第 1 章 MySQL 基本介绍

前言：

作为最为流行的开源数据库软件之一，MySQL 数据库软件已经是广为人知了。但是为了照顾对 MySQL 还不熟悉的读者，这章我们将对 MySQL 做一个简单的介绍。主要内容包括 MySQL 各功能模块组成，各模块协同工作原理，Query 处理的流程等。

1．1 MySQL Server 简介

1.1.1 什么是 MySQL

MySQL 是由 MySQL AB 公司（目前已经被 SUN 公司收归麾下）自主研发的，目前 IT 行业最流行的开放源代码的数据库管理系统之一，它同时也是一个支持多线程高并发多用户的关系型数据库管理系统。

MySQL 数据库以其简单高效可靠的特点，在最近短短几年的时间就从一个名不见经传的数据库系统，变成一个在 IT 行业几乎是无人不知的开源数据库管理系统。从微型的嵌入式系统，到小型的 web 网站，至大型的企业级应用，到处都可见其身影的存在。为何一个开源的数据库管理系统会变得如此的流行呢？在我 2003 年第一次接触 MySQL 之前，也是非常的纳闷？或许在我大概的向您介绍一下其发展历程之后，心中的这个问题就会消失了。

1.1.2 艰难诞生

1985 年，瑞典的几位志同道合小伙子（以 David Axmark 为首） 成立了一家公司，这就是 MySQL AB 的前身。这个公司最初并不是为了开发数据库产品，而是在实现他们想法的过程中，需要一个数据库。他们希望能够使用开源的产品。但在当时并没有一个合适的选择，没办法，那就自己开发吧。

在最初，他们只是自己设计了一个利用索引顺序存取数据的方法，也就是 ISAM（Indexed Sequential Access Method）存储引擎核心算法的前身，利用 ISAM 结合 mSQL 来实现他们的应用需求。在早期，他们主要是为瑞典的一些大型零售商提供数据仓库服务。在系统使用过程中，随着数据量越来越大，系统复杂度越来越高，ISAM 和 mSQL 的组合逐渐不堪重负。在分析性能瓶颈之后，他们发现问题出在 mSQL 上面。不得已，他们抛弃了 mSQL，重新开发了一套功能类似的数据存储引擎，这就是 ISAM 存储引擎。大家可能已经注意到他们当时的主要客户是数据仓库，应该也容易理解为什么直至现在，MySQL 最擅长的是查询性能，而不是

VISIT…

LANZAROTE
Caliente.COM

事务处理（需要借助第三方存储引擎)。

软件诞生，自然该给她取一个好听并且容易记住的名字。时至今日，MySQL AB 仍然没有公布当初为什么给这个数据库系统取名为 MySQL。据传 MySQL 是取自创始人之一 Monty Widenius 的女儿的名字或许大家会认为这仅仅是我的猜测，不以为然，其实也并不是完全没有根据的。大家或许知道 MySQL 最近正在研发的用来替代 MyISAM 存储引擎的新一代存储引擎产品 Maria,为什么叫 Maria? 笔者对这个问题也比较感兴趣,曾经和 MySQL 前 CTO David 沟通过。得到的答案是，Maria 是以他小女儿的名字命名的。看来，这是几位 MySQL 的创始人为自己的软件命名的一个习惯。

在 MySQL 诞生之初，其功能还非常粗糙，和当时已经成熟稳定运营多年的商业数据库管理系统完全不能比。MySQL 之所以能够成功，和几个创始人最初采用的策略关系非常大。

1.1.3 寻求发展

MySQL 诞生的时候，正是互联网开始高速发展的时期。MySQL AB 通过优化 MySQL 满足了互联网开发用者对数据库产品的需求：标准化查询语言的支持，高效的数据存取，不必关注事务完整性，简单易用，而且成本低廉。当时大量的小公司都愿意采用 MySQL 作为数据库应用系统的数据库管理系统，所以 MySQL 的用户数量不断的增长，进一步促进了 MySQL 自身的不断改进和完善，进入了一个非常好的良性循环。

合理地把握需求， 准确地定位目标客户，为 MySQL 后面的发展铺平了道路。我们看到，MySQL 一开始就没有拿大型的企业管理软件的数据库系统来定位自己，没有将所有的 IT 行业定位为自己的目标用户，而是选择的当时并不受重视的一小部分 Web 开发者作为自己的客户来重点培养发展。这种做法或许值得我们的 IT 企业学习。

1.1.4 巨人崛起

可以说，正是 MySQL 最初抓住了互联网客户，造就了今天 MySQL 在互联网行业的巨大成功。当然，MySQL 的高速发展，同时也离不开另外一个很关键的因素，那就是开放源代码。

在 2000 年的时候，MySQL 公布了自己的源代码，并采用 GPL（GNU General Public License）许可协议，正式进入开源世界。虽然在当时的环境下，开源还没有现在这样流行，但是那是开源世界开始真正让大多数世人所接受并开始推崇的起步阶段。当然 MySQL 的成功并不仅仅是因为以上的这些原因，但我们不能否认正是 MySQL 这一战略性质的策略让 MySQL 在进一步拓展自己的客户群 的路上一路东风。此后 MySQL 的发展路程我想就不需要我继续再次罗嗦了，因为基本上都可以从 MySQL 的官方网站（http://www.mysql.com）得到相应的答案。

1．2 MySQL 与其他数据库的简单比较

前面我们简单介绍了 MySQL 的发展历程，从中了解了 MySQL 快速崛起的必要的条件。接下来，我们通过 MySQL 在功能，性能，以及其易用性方面和其他主流的数据库做一个基本的比较，来了解一下 MySQL 成为当下最流行的开源数据库软件的充分条件。

1.2.1 功能比较

作为一个成熟的数据库管理系统，要满足各种各样的商业需求，功能肯定是会被列入重点参考对象的。MySQL 虽然在早期版本的时候功能非常简单，只能做一些很基础的结构化数据存取操作，但是经过多年的改进和完善之后，已经基本具备了所有通用数据库管理系统所需要的相关功能。

MySQL 基本实现了 ANSI SQL 92 的大部分标准，仅有少部分并不经常被使用的部分没有实现。比如在字段类型支持方面，另一个著名的开源数据库 PostGreSQL 支持的类型是最完整的，而 Oracle 和其他一些商业数据库，比如 DB2、Sybase 等，较 MySQL 来说也要相对少一些。这一点，我们可以通过 TCX 的 crash-me 测试套件所得出的测试报告得知。在事务支持方面，虽然 MySQL 自己的存储引擎并没有提供，但是已经通过第三方插件式存储引擎 Innodb 实现了 SQL 92 标准所定义的四个事务隔离级别的全部，只是在实现的过程中每一种的实现方式可能有一定的区别，这在当前商用数据库管理系统中都不多见。比如，大家所熟知的大名鼎鼎的Oracle数据库就仅仅实现了其中的两种(Serializable和Read Commited)，而 PostGreSQL，同样支持四种隔离级别。

不过在可编程支持方面，MySQL 和其他数据库相比还有一定的差距，虽然最新版的 MySQL 已经开始提供一些简单的可编程支持，如开始支持 Procedure，Function，Trigger 等，但是所支持的功能还比较有限，和其他几大商用数据库管理系统相比，还存在较大的不足。如 Oracle 有强大的 PL/SQL，SQL Server 有 T-SQL，PostGreSQL 也有功能很完善的 PL/PGSQL 的支持。

整体来说，虽然在功能方面 MySQL 数据库作为一个通用的数据库管理系统暂时还无法和 PostGreSQL 相比，但是其功能完全可以满足我们的通用商业需求，提供足够强大的服务。而且不管是哪一种数据库在功能方面都不敢声称自己比其他任何一款商用通用数据库管理系统都强，甚至都不敢声称能够自己拥有某一数据库产品的所有功能。因为每一款数据库管理系统都有起自身的优势，也有起自身的限制，这只能代表每一款产品所倾向的方向不一样而已。

1.2.2 易用性比较

从系统易用性方面来比较，每一个使用过 MySQL 的用户都能够明显地感觉出 MySQL 在这

方面与其他通用数据库管理系统之间的优势所在。尤其是相对于一些大型的商业数据库管理系统如 Oracle、DB2 以及 Sybase 来说，对于普通用户来说，操作的难易程度明显不处于一个级别。MySQL 一直都奉行简单易用的原则，也正是靠这一特性，吸引了大量的初级数据库用户最终选择了 MySQL。也正是这一批又一批的初级用户，在经过了几年时间的成长之后，很多都已经成为了高级数据库用户，而且也一直都在伴随着 MySQL 成长。

从安装方面来说，MySQL 安装包大小仅仅只有 100MB 左右，这几大商业数据库完全不在一个数量级。安装难易程度也要比 Oracle 等商业数据库简单很多，不论是通过已经编译好的二进制分发包还是源码编译安装，都非常简单。

再从数据库创建来比较，MySQL 仅仅只需要一个简单的 CREATE DATABASE 命令，即可在瞬间完成建库的动作，而 Oracle 数据库与之相比，创建一个数据库简直就是一个非常庞大的工程。当然，二者数据库的概念存在一定差别也不可否认。

1.2.3 性能比较

性能方面，一直是 MySQL 引以为自豪的一个特点。在权威的第三方评测机构多次测试较量各种数据库 TPCC 值的过程中，MySQL 一直都有非常优异的表现，而且在其他所有商用的通用数据库管理系统中，仅仅只有 Oracle 数据库能够与其一较高下。至于各种数据库详细的性能数据，我这里就不便记录，大家完全可以通过网上第三方评测机构公布的数据了解具体细节信息。

MySQL 一直以来奉行一个原则，那就是在保证足够的稳定性的前提下，尽可能的提高自身的处理能力。也就是说，在性能和功能方面，MySQL 第一考虑的要素主要还是性能，MySQL 希望自己是一个在满足客户 99%的功能需求的前提下，花掉剩下的大部分精力来性能努力，而不是希望自己是成为一个比其他任何数据库的功能都要强大的数据库产品。

1.2.4 可靠性

关于可靠性的比较，并没有太多详细的评测比较数据，但是从目前业界的交流中可以了解到，几大商业厂商的数据库的可靠性肯定是没有太多值得怀疑的。但是做为开源数据库管理系统的代表，MySQL 也有非常优异的表现，而并不是像有些人心中所怀疑的那样，因为不是商业厂商所提供，就会不够稳定不够健壮。从当前最火的 Facebook 这样大型的网站都是使用 MySQL 数据库，就可以看出，MySQL 在稳定可靠性方面，并不会比我们的商业厂商的产品有太多逊色。而且排在全球前 10 位的大型网站里面，大部分都有部分业务是运行在 MySQL 数据库环境上，如 Yahoo，Google 等。

总的来说，MySQL 数据库在发展过程中一直有自己的三个原则：简单、高效、可靠。从上面的简单比较中，我们也可以看出，在 MySQL 自己的所有三个原则上面，没有哪一项是做得不好的。而且，虽然功能并不是 MySQL 自身所追求的三个原则之一，但是考虑到当前用户量的急剧增长，用户需求越来越越多样化，MySQL 也不得不在功能方面做出大量的努力，来

不断满足客户的新需求。比如最近版本中出现的Eent Scheduler（类似于Oracle的Job功能），Partition功能，自主研发的Maria存储引擎在功能方面的扩展，Falcon存储引擎对事务的支持等等，都证明了MySQL在功能方面也开始了不懈的努力。

任何一种产品，都不可能是完美的，也不可能适用于所有用户。我们只有衡量了每一种产品的各种特性之后，从中选择出一种最适合于自身的产品。

1．3 MySQL 的主要适用场景

据说目前MySQL用户已经达千万级别了，其中不乏企业级用户。可以说是目前最为流行的开源数据库管理系统软件了。任何产品都不可能是万能的，也不可能适用于所有的应用场景。那么MySQL到底在什么场景下适用什么场景下不适用呢？

1、Web网站系统

Web站点，是MySQL最大的客户群，也是MySQL发展史上最为重要的支撑力量，这一点在最开始的MySQL Server简介部分就已经说明过。

MySQL之所以能成为Web站点开发者们最青睐的数据库管理系统，是因为MySQL数据库的安装配置都非常简单，使用过程中的维护也不像很多大型商业数据库管理系统那么复杂，而且性能出色。还有一个非常重要的原因就是MySQL是开放源代码的，完全可以免费使用。

2、日志记录系统

MySQL数据库的插入和查询性能都非常的高效，如果设计地较好，在使用 MyISAM存储引擎的时候，两者可以做到互不锁定，达到很高的并发性能。所以，对需要大量的插入和查询日志记录的系统来说，MySQL是非常不错的选择。比如处理用户的登录日志，操作日志等，都是非常适合的应用场景。

3、数据仓库系统

随着现在数据仓库数据量的飞速增长，我们需要的存储空间越来越大。数据量的不断增长，使数据的统计分析变得越来越低效，也越来越困难。怎么办？这里有几个主要的解决思路，一个是采用昂贵的高性能主机以提高计算性能，用高端存储设备提高I/O性能，效果理想，但是成本非常高；第二个就是通过将数据复制到多台使用大容量硬盘的廉价pc server上，以提高整体计算性能和I/O能力，效果尚可，存储空间有一定限制，成本低廉；第三个，通过将数据水平拆分，使用多台廉价的pc server和本地磁盘来存放数据，每台机器上面都只有所有数据的一部分，解决了数据量的问题，所有pc server一起并行计算，也解决了计算能力问题，通过中间代理程序调配各台机器的运算任务，既可以解决计算性能问题又可以解决I/O性能问题，成本也很低廉。在上面的三个方案中，第二和第三个的实现，MySQL都有较大的优势。通过MySQL的简单复制功能，可以很好的将数据从一台主机复制到另外一台，不仅仅在局域网内可以复制，在广域网同样可以。当然，很多人可能会说，其他的数据库同样也可以做到，不是只有 MySQL有这样的功能。确实，很多数据库同样能做到，但是MySQL是免费的，其他数据库大多都是按照主机数量或者cpu数量来收费，当我们使用大量的pc server的时候，license费用相当惊人。第一个方案，基本上所有数据库系统都能够实现，

但是其高昂的成本并不是每一个公司都能够承担的。

4、嵌入式系统

嵌入式环境对软件系统最大的限制是硬件资源非常有限，在嵌入式环境下运行的软件系统，必须是轻量级低消耗的软件。

MySQL 在资源的使用方面的伸缩性非常大，可以在资源非常充裕的环境下运行，也可以在资源非常少的环境下正常运行。它对于嵌入式环境来说，是一种非常合适的数据库系统，而且 MySQL 有专门针对于嵌入式环境的版本。

1．4 小结

从最初的诞生，到发展成为目前最为流行的开源数据库管理软件，MySQL 已经走过了较长的一段路，也正是这段不寻常的路，造就了今天 MySQL 的成就。

通过本章的信息，我想各位读者应该是比较清楚 MySQL 的大部分基本信息了，对于 MySQL 主要特长，以及适用场景，都有了一个初步的了解。在后续章节中我们将会针对这些内容做更为详细深入的介绍。

第 2 章 MySQL 架构组成

前言

麻雀虽小，五脏俱全。MySQL 虽然以简单著称，但其内部结构并不简单。本章从 MySQL 物理组成、逻辑组成，以及相关工具几个角度来介绍 MySQL 的整体架构组成，希望能够让读者对 MySQL 有一个更全面深入的了解。

2．1 MySQL 物理文件组成

2.1.1 日志文件

1、错误日志：Error Log

错误日志记录了 MyQL Server 运行过程中所有较为严重的警告和错误信息，以及 MySQL Server 每次启动和关闭的详细信息。在默认情况下，系统记录错误日志的功能是关闭的，错误信息被输出到标准错误输出（stderr），如果要开启系统记录错误日志的功能，需要在启动时开启-log-error 选项。错误日志的默认存放位置在数据目录下，以 hostname.err 命名。但是可以使用命令：--log-error[=file_name]，修改其存放目录和文件名。

为了方便维护需要，有时候会希望将错误日志中的内容做备份并重新开始记录，这时候就可以利用 MySQL 的 FLUSH LOGS 命令来告诉 MySQL 备份旧日志文件并生成新的日志文件。备份文件名以“.old”结尾。

2、二进制日志：Binary Log & Binary Log Index

二进制日志，也就是我们常说的 binlog，也是 MySQL Server 中最为重要的日志之一。当我们通过“--log-bin[=file_name]”打开了记录的功能之后，MySQL 会将所有修改数据库数据的 query 以二进制形式记录到日志文件中。当然，日志中并不仅限于 query 语句这么简单，还包括每一条 query 所执行的时间，所消耗的资源，以及相关的事务信息，所以 binlog 是事务安全的。

和错误日志一样，binlog 记录功能同样需要“--log-bin[=file_name]”参数的显式指定才能开启，如果未指定 file_name，则会在数据目录下记录为 mysql-bin.******(*代表0～9 之间的某一个数字，来表示该日志的序号)。

binlog 还有其他一些附加选项参数：

“--max_binlog_size”设置 binlog 的最大存储上限，当日志达到该上限时，MySQL 会重新创建一个日志开始继续记录。不过偶尔也有超出该设置的 binlog 产生，一般都是因为在即将达到上限时，产生了一个较大的事务，为了保证事务安全，MySQL 不会将同一个事务分开记录到两个 binlog 中。

“--binlog-do-db=db_name”参数明确告诉 MySQL，需要对某个（db_name）数据库记录 binlog，如果有了“--binlog-do-db=db_name”参数的显式指定，MySQL 会忽略针对其他数据库执行的 query，而仅仅记录针对指定数据库执行的 query。

“--binlog-ignore-db=db_name”与“--binlog-do-db=db_name”完全相反，它显式指定忽略某个（db_name）数据库的 binlog 记录，当指定了这个参数之后，MySQL 会记录指定数据库以外所有的数据库的 binlog。

“--binlog-ignore-db=db_name”与“--binlog-do-db=db_name”两个参数有一个共同的概念需要大家理解清楚，参数中的 db_name 不是指 query 语句更新的数据所在的数据库，而是执行 query 的时候当前所处的数据库。不论更新哪个数据库的数据，MySQL 仅仅比较当前连接所处的数据库（通过 use db_name 切换后所在的数据库）与参数设置的数据库名，而不会分析 query 语句所更新数据所在的数据库。

mysql-bin.index 文件（binary log index）的功能是记录所有 Binary Log 的绝对路径，保证 MySQL 各种线程能够顺利的根据它找到所有需要的 Binary Log 文件。

3、更新日志：update log

更新日志是 MySQL 在较老的版本上使用的，其功能和 binlog 基本类似，只不过不是以二进制格式来记录而是以简单的文本格式记录内容。自从 MySQL 增加了 binlog 功能之后，就很少使用更新日志了。从版本 5.0 开始，MySQL 已经不再支持更新日志了。

4、查询日志：query log

查询日志记录 MySQL 中所有的 query，通过"--log[=fina_name]"来打开该功能。由于记录了所有的 query，包括所有的 select，体积比较大，开启后对性能也有较大的影响，所以请大家慎用该功能。一般只用于跟踪某些特殊的 sql 性能问题才会短暂打开该功能。默认的查询日志文件名为 hostname.log。

5、慢查询日志：slow query log

顾名思义，慢查询日志中记录的是执行时间较长的 query，也就是我们常说的 slow query，通过设--log-slow-queries[=file_name]来打开该功能并设置记录位置和文件名，默认文件名为 hostname-slow.log，默认目录也是数据目录。

慢查询日志采用的是简单的文本格式，可以通过各种文本编辑器查看其中的内容。其中记录了语句执行的时刻，执行所消耗的时间，执行用户，连接主机等相关信息。MySQL 还提供了专门用来分析满查询日志的工具程序 mysqlslowdump，用来帮助数据库管理人员解决可能存在的性能问题。

6、Innodb 的在线 redo 日志：innodb redo log

Innodb 是一个事务安全的存储引擎，其事务安全性主要就是通过在线 redo 日志和记录在表空间中的 undo 信息来保证的。redo 日志中记录了 Innodb 所做的所有物理变更和事务信息，通过 redo 日志和 undo 信息，Innodb 保证了在任何情况下的事务安全性。Innodb 的 redo 日志同样默认存放在数据目录下，可以通过 innodb_log_group_home_dir 来更改设置日志的存放位置，通过 innodb_log_files_in_group 设置日志的数量。

2.1.2 数据文件

在 MySQL 中每一个数据库都会在定义好（或者默认）的数据目录下存在一个以数据库名字命名的文件夹，用来存放该数据库中各种表数据文件。不同的 MySQL 存储引擎有各自不同的数据文件，存放位置也有区别。多数存储引擎的数据文件都存放在和 MyISAM 数据文件位置相同的目录下，但是每个数据文件的扩展名却各不一样。如 MyISAM 用".MYD"作为扩展名，Innodb 用".ibd"，Archive 用".arc"，CSV 用".csv"，等等。

1、".frm"文件

与表相关的元数据（meta）信息都存放在".frm"文件中，包括表结构的定义信息等。不论是什么存储引擎，每一个表都会有一个以表名命名的".frm"文件。所有的".frm"文件都存放在所属数据库的文件夹下面。

2、".MYD"文件

“.MYD”文件是MyISAM存储引擎专用，存放MyISAM表的数据。每一个MyISAM表都会有一个“.MYD”文件与之对应，同样存放于所属数据库的文件夹下，和“.frm”文件在一起。

3、“.MYI”文件

“.MYI”文件也是专属于MyISAM存储引擎的，主要存放MyISAM表的索引相关信息。对于MyISAM存储来说，可以被cache的内容主要就是来源于“.MYI”文件中。每一个MyISAM表对应一个“.MYI”文件，存放于位置和“.frm”以及“.MYD”一样。

4、“.ibd”文件和ibdata文件

这两种文件都是存放Innodb数据的文件，之所以有两种文件来存放Innodb的数据（包括索引），是因为Innodb的数据存储方式能够通过配置来决定是使用共享表空间存放存储数据，还是独享表空间存放存储数据。独享表空间存储方式使用“.ibd”文件来存放数据，且每个表一个“.ibd”文件，文件存放在和MyISAM数据相同的位置。如果选用共享存储表空间来存放数据，则会使用ibdata文件来存放，所有表共同使用一个（或者多个，可自行配置）ibdata文件。ibdata文件可以通过innodb_data_home_dir和innodb_data_file_path两个参数共同配置组成，innodb_data_home_dir配置数据存放的总目录，而innodb_data_file_path配置每一个文件的名称。当然，也可以不配置innodb_data_home_dir而直接在innodb_data_file_path参数配置的时候使用绝对路径来完成配置。innodb_data_file_path中可以一次配置多个ibdata文件。文件可以是指定大小，也可以是自动扩展的，但是Innodb限制了仅仅只有最后一个ibdata文件能够配置成自动扩展类型。当我们需要添加新的ibdata文件的时候，只能添加在innodb_data_file_path配置的最后，而且必须重启MySQL才能完成ibdata的添加工作。不过如果我们使用独享表空间存储方式的话，就不会有这样的问题，但是如果要使用裸设备的话，每个表一个裸设备，可能造成裸设备数量非常大，而且不太容易控制大小，实现比较困难，而共享表空间却不会有这个问题，容易控制裸设备数量。我个人还是更倾向于使用独享表空间存储方式。当然，两种方式各有利弊，看大家各自应用环境的侧重点在那里了。

上面仅仅介绍了两种最常用存储引擎的数据文件，此外其他各种存储引擎都有各自的数据文件，读者朋友可以自行创建某个存储引擎的表做一个简单的测试，做更多的了解。

2.1.3 Replication 相关文件：

1、master.info文件：

master.info文件存在于Slave端的数据目录下，里面存放了该Slave的Master端的相关信息，包括Master的主机地址，连接用户，连接密码，连接端口，当前日志位置，已经读取到的日志位置等信息。

2、relay log 和relay log index

mysql-relay-bin.xxxxxn 文件用于存放 Slave 端的 I/O 线程从 Master 端所读取到的 Binary Log 信息，然后由 Slave 端的 SQL 线程从该 relay log 中读取并解析相应的日志信息，转化成 Master 所执行的 SQL 语句，然后在 Slave 端应用。

mysql-relay-bin.index 文件的功能类似于 mysql-bin.index ，同样是记录日志的存

放位置的绝对路径，只不过他所记录的不是 Binary Log，而是 Relay Log。

3、relay-log.info 文件：

类似于 master.info，它存放通过 Slave 的 I/O 线程写入到本地的 relay log 的相关信息。供 Slave 端的 SQL 线程以及某些管理操作随时能够获取当前复制的相关信息。

2.1.4 其他文件：

1、system config file

MySQL 的系统配置文件一般都是“my.cnf”，Unix/Linux 下默认存放在"/etc"目录下，Windows 环境一般存放在“c:/windows”目录下面。“my.cnf”文件中包含多种参数选项组（group），每一种参数组都通过中括号给定了固定的组名，如“[mysqld]”组中包括了 mysqld 服务启动时候的初始化参数，“[client]”组中包含着客户端工具程序可以读取的参数，此外还有其他针对于各个客户端软件的特定参数组，如 mysql 程序使用的“[mysql]”，mysqlchk 使用的“[mysqlchk]”，等等。如果读者朋友自己编写了某个客户端程序，也可以自己设定一个参数组名，将相关参数配置在里面，然后调用 mysql 客户端 api 程序中的参数读取 api 读取相关参数。

2、pid file

pid file 是 mysqld 应用程序在 Unix/Linux 环境下的一个进程文件，和许多其他 Unix/Linux 服务端程序一样，存放着自己的进程 id。

3、socket file

socket 文件也是在 Unix/Linux 环境下才有的，用户在 Unix/Linux 环境下客户端连接可以不通过 TCP/IP 网络而直接使用 Unix Socket 来连接 MySQL。

2．2 MySQL Server 系统架构

2.2.1 逻辑模块组成

总的来说，MySQL 可以看成是二层架构，第一层我们通常叫做 SQL Layer，在 MySQL 数据库系统处理底层数据之前的所有工作都是在这一层完成的，包括权限判断，sql 解析，执行计划优化，query cache 的处理等等；第二层就是存储引擎层，我们通常叫做 Storage Engine Layer，也就是底层数据存取操作实现部分，由多种存储引擎共同组成。所以，可以用如下一张最简单的架构示意图来表示 MySQL 的基本架构，如图 2-1 所示：

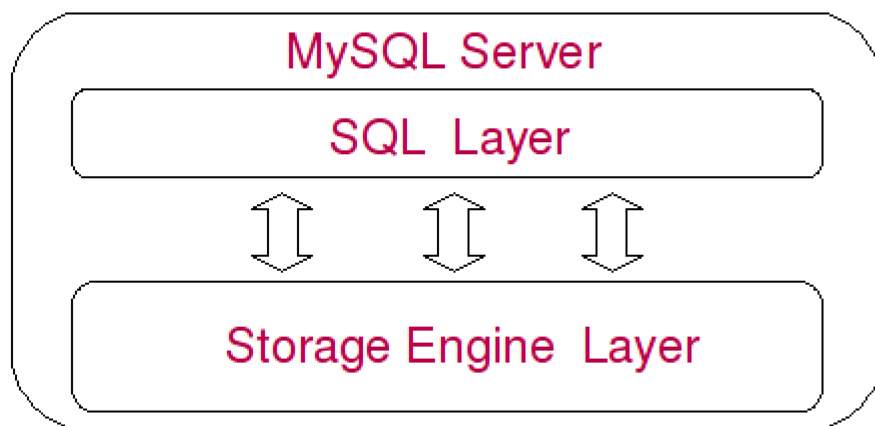


图 2-1

虽然从上图看起来 MySQL 架构非常的简单，就是简单的两部分而已，但实际上每一层中都含有各自的很多小模块，尤其是第一层 SQL Layer，结构相当复杂的。下面我们就分别针对 SQL Layer 和 Storage Engine Layer 做一个简单的分析。

SQL Layer 中包含了多个子模块，下面我将逐个做一下简单的介绍：

1、初始化模块

顾名思议，初始化模块就是在 MySQL Server 启动的时候，对整个系统做各种各样的初始化操作，比如各种 buffer，cache 结构的初始化和内存空间的申请，各种系统变量的初始化设定，各种存储引擎的初始化设置，等等。

2、核心 API

核心 API 模块主要是为了提供一些需要非常高效的底层操作功能的优化实现，包括各种底层数据结构的实现，特殊算法的实现，字符串处理，数字处理等，小文件 I/O，格式化输出，以及最重要的内存管理部分。核心 API 模块的所有源代码都集中在 mysys 和 strings 文件夹下面，有兴趣的读者可以研究研究。

3、网络交互模块

底层网络交互模块抽象出底层网络交互所使用的接口 api，实现底层网络数据的接收与发送，以方便其他各个模块调用，以及对这一部分的维护。所有源码都在 vio 文件夹下面。

4、Client & Server 交互协议模块

任何 C/S 结构的软件系统，都肯定会有自己独有的信息交互协议，MySQL 也不例外。MySQL 的 Client & Server 交互协议模块部分，实现了客户端与 MySQL 交互过程中的所有协议。当然这些协议都是建立在现有的 OS 和网络协议之上的，如 TCP/IP 以及 Unix Socket。

5、用户模块

用户模块所实现的功能，主要包括用户的登录连接权限控制和用户的授权管理。他就像 MySQL 的大门守卫一样，决定是否给来访者“开门”。

6、访问控制模块

造访客人进门了就可以想干嘛就干嘛么？为了安全考虑，肯定不能如此随意。这时候就需要访问控制模块实时监控客人的每一个动作，给不同的客人以不同的权限。访问控制模块实现的功能就是根据用户模块中各用户的授权信息，以及数据库自身特有的各种约束，来控制用户对数据的访问。用户模块和访问控制模块两者结合起来，组成了 MySQL 整个数据库系统的权限安全管理的功能。

7、连接管理、连接线程和线程管理

连接管理模块负责监听对 MySQL Server 的各种请求，接收连接请求，转发所有连接请求到线程管理模块。每一个连接上 MySQL Server 的客户端请求都会被分配（或创建）一个连接线程为其单独服务。而连接线程的主要工作就是负责 MySQL Server 与客户端的通信，接受客户端的命令请求，传递 Server 端的结果信息等。线程管理模块则负责管理维护这些连接线程。包括线程的创建，线程的 cache 等。

8、Query 解析和转发模块

在 MySQL 中我们习惯将所有 Client 端发送给 Server 端的命令都称为 query，在 MySQL Server 里面，连接线程接收到客户端的一个 Query 后，会直接将该 query 传递给专门负责将各种 Query 进行分类然后转发给各个对应的处理模块，这个模块就是 query 解析和转发模块。其主要工作就是将 query 语句进行语义和语法的分析，然后按照不同的操作类型进行分类，然后做出针对性的转发。

9、Query Cache 模块

Query Cache 模块在 MySQL 中是一个非常重要的模块，他的主要功能是将客户端提交给 MySQL 的 Select 类 query 请求的返回结果集 cache 到内存中，与该 query 的一个 hash 值做一个对应。该 Query 所取数据的基表发生任何数据的变化之后，MySQL 会自动使该 query 的 Cache 失效。在读写比例非常高的应用系统中，Query Cache 对性能的提高是非常显著的。当然它对内存的消耗也是非常大的。

10、Query 优化器模块

Query 优化器，顾名思义，就是优化客户端请求的 query，根据客户端请求的 query 语句，和数据库中的一些统计信息，在一系列算法的基础上进行分析，得出一个最优的策略，告诉后面的程序如何取得这个 query 语句的结果。

11、表变更管理模块

表变更管理模块主要是负责完成一些 DML 和 DDL 的 query，如：update，delte，insert，create table，alter table 等语句的处理。

12、表维护模块

表的状态检查，错误修复，以及优化和分析等工作都是表维护模块需要做的事情。

13、系统状态管理模块

系统状态管理模块负责在客户端请求系统状态的时候，将各种状态数据返回给用户，像 DBA 常用的各种 show status 命令，show variables 命令等，所得到的结果都是由这个模块返回的。

14、表管理器

这个模块从名字上看来很容易和上面的表变更和表维护模块相混淆，但是其功能与变更及维护模块却完全不同。大家知道，每一个 MySQL 的表都有一个表的定义文件，也就是*.frm 文件。表管理器的工作主要就是维护这些文件，以及一个 cache，该 cache 中的主要内容是各个表的结构信息。此外它还维护 table 级别的锁管理。

15、日志记录模块

日志记录模块主要负责整个系统级别的逻辑层的日志的记录，包括 error log，binary log，slow query log 等。

16、复制模块

复制模块又可分为 Master 模块和 Slave 模块两部分，Master 模块主要负责在 Replication 环境中读取 Master 端的 binary 日志，以及与 Slave 端的 I/O 线程交互等工作。Slave 模块比 Master 模块所要做的事情稍多一些，在系统中主要体现在两个线程上面。一个是负责从 Master 请求和接受 binary 日志，并写入本地 relay log 中的 I/O 线程。另外一个是负责从 relay log 中读取相关日志事件，然后解析成可以在 Slave 端正确执行并得到和 Master 端完全相同的结果的命令并再交给 Slave 执行的 SQL 线程。

17、存储引擎接口模块

存储引擎接口模块可以说是 MySQL 数据库中最有特色的一点了。目前各种数据库产品中，基本上只有 MySQL 可以实现其底层数据存储引擎的插件式管理。这个模块实际上只是一个抽象类，但正是因为它成功地将各种数据处理高度抽象化，才成就了今天 MySQL 可插拔存储引擎的特色。

2.2.2 各模块工作配合

在了解了 MySQL 的各个模块之后，我们再看看 MySQL 各个模块间是如何相互协同工作的。接下来，我们通过启动 MySQL，客户端连接，请求 query，得到返回结果，最后退出，这样一整个过程来进行分析。

当我们执行启动 MySQL 命令之后，MySQL 的初始化模块就从系统配置文件中读取系统参数和命令行参数，并按照参数来初始化整个系统，如申请并分配 buffer，初始化全局变量，以及各种结构等。同时各个存储引擎也被启动，并进行各自的初始化工作。当整个系统初始化结束后，由连接管理模块接手。连接管理模块会启动处理客户端连接请求的监听程序，包括 tcp/ip 的网络监听，还有 unix 的 socket。这时候，MySQL Server 就基本启动完成，准备好接受客户端请求了。

当连接管理模块监听到客户端的连接请求（借助网络交互模块的相关功能），双方通过 Client & Server 交互协议模块所定义的协议“寒暄”几句之后，连接管理模块就会将连接请求转发给线程管理模块，去请求一个连接线程。

线程管理模块马上又会将控制交给连接线程模块，告诉连接线程模块：现在我这边有连

接请求过来了，需要建立连接，你赶快处理一下。连接线程模块在接到连接请求后，首先会检查当前连接线程池中是否有被cache的空闲连接线程，如果有，就取出一个和客户端请求连接上，如果没有空闲的连接线程，则建立一个新的连接线程与客户端请求连接。当然，连接线程模块并不是在收到连接请求后马上就会取出一个连接线程连和客户端连接，而是首先通过调用用户模块进行授权检查，只有客户端请求通过了授权检查后，他才会将客户端请求和负责请求的连接线程连上。

在MySQL中，将客户端请求分为了两种类型：一种是query，需要调用Parser也就是Query解析和转发模块的解析才能够执行的请求；一种是command，不需要调用Parser就可以直接执行的请求。如果我们的初始化配置中打开了Full Query Logging的功能，那么Query解析与转发模块会调用日志记录模块将请求计入日志，不管是一个Query类型的请求还是一个command类型的请求，都会被记录进入日志，所以出于性能考虑，一般很少打开Full Query Logging的功能。

当客户端请求和连接线程“互换暗号（互通协议）”接上头之后，连接线程就开始处理客户端请求发送过来的各种命令（或者query），接受相关请求。它将收到的query语句转给Query解析和转发模块，Query解析器先对Query进行基本的语义和语法解析，然后根据命令类型的不同，有些会直接处理，有些会分发给其他模块来处理。

如果是一个Query类型的请求，会将控制权交给Query解析器。Query解析器首先分析看是不是一个select类型的query，如果是，则调用查询缓存模块，让它检查该query在query cache中是否已经存在。如果有，则直接将cache中的数据返回给连接线程模块，然后通过与客户端的连接的线程将数据传输给客户端。如果不是一个可以被cache的query类型，或者cache中没有该query的数据，那么query将被继续传回query解析器，让query解析器进行相应处理，再通过query分发器分发给相关处理模块。

如果解析器解析结果是一条未被cache的select语句，则将控制权交给Optimizer，也就是Query优化器模块，如果是DML或者是DDL语句，则会交给表变更管理模块，如果是一些更新统计信息、检测、修复和整理类的query则会交给表维护模块去处理，复制相关的query则转交给复制模块去进行相应的处理，请求状态的query则转交给了状态收集报告模块。实际上表变更管理模块根据所对应的处理请求的不同，是分别由insert处理器、delete处理器、update处理器、create处理器，以及alter处理器这些小模块来负责不同的DML和DDL的。

在各个模块收到Query解析与分发模块分发过来的请求后，首先会通过访问控制模块检查连接用户是否有访问目标表以及目标字段的权限，如果有，就会调用表管理模块请求相应的表，并获取对应的锁。表管理模块首先会查看该表是否已经存在于table cache中，如果已经打开则直接进行锁相关的处理，如果没有在cache中，则需要再打开表文件获取锁，然后将打开的表交给表变更管理模块。

当表变更管理模块“获取”打开的表之后，就会根据该表的相关meta信息，判断表的存储引擎类型和其他相关信息。根据表的存储引擎类型，提交请求给存储引擎接口模块，调用对应的存储引擎实现模块，进行相应处理。

不过，对于表变更管理模块来说，可见的仅是存储引擎接口模块所提供的一系列“标准”接口，底层存储引擎实现模块的具体实现，对于表变更管理模块来说是透明的。他只需要调用对应的接口，并指明表类型，接口模块会根据表类型调用正确的存储引擎来进行相应的处理。

当一条 query 或者一个 command 处理完成（成功或者失败）之后，控制权都会交还给连接线程模块。如果处理成功，则将处理结果（可能是一个 Result set，也可能是成功或者失败的标识）通过连接线程反馈给客户端。如果处理过程中发生错误，也会将相应的错误信息发送给客户端，然后连接线程模块会进行相应的清理工作，并继续等待后面的请求，重复上面提到的过程，或者完成客户端断开连接的请求。

如果在上面的过程中，相关模块使数据库中的数据发生了变化，而且 MySQL 打开了 bin-log 功能，则对应的处理模块还会调用日志处理模块将相应的变更语句以更新事件的形式记录到相关参数指定的二进制日志文件中。

在上面各个模块的处理过程中，各自的核心运算处理功能部分都会高度依赖整个 MySQL 的核心 API 模块，比如内存管理，文件 I/O，数字和字符串处理等等。

了解到整个处理过程之后，我们可以将以上各个模块画成如图 2-2 的关系图：

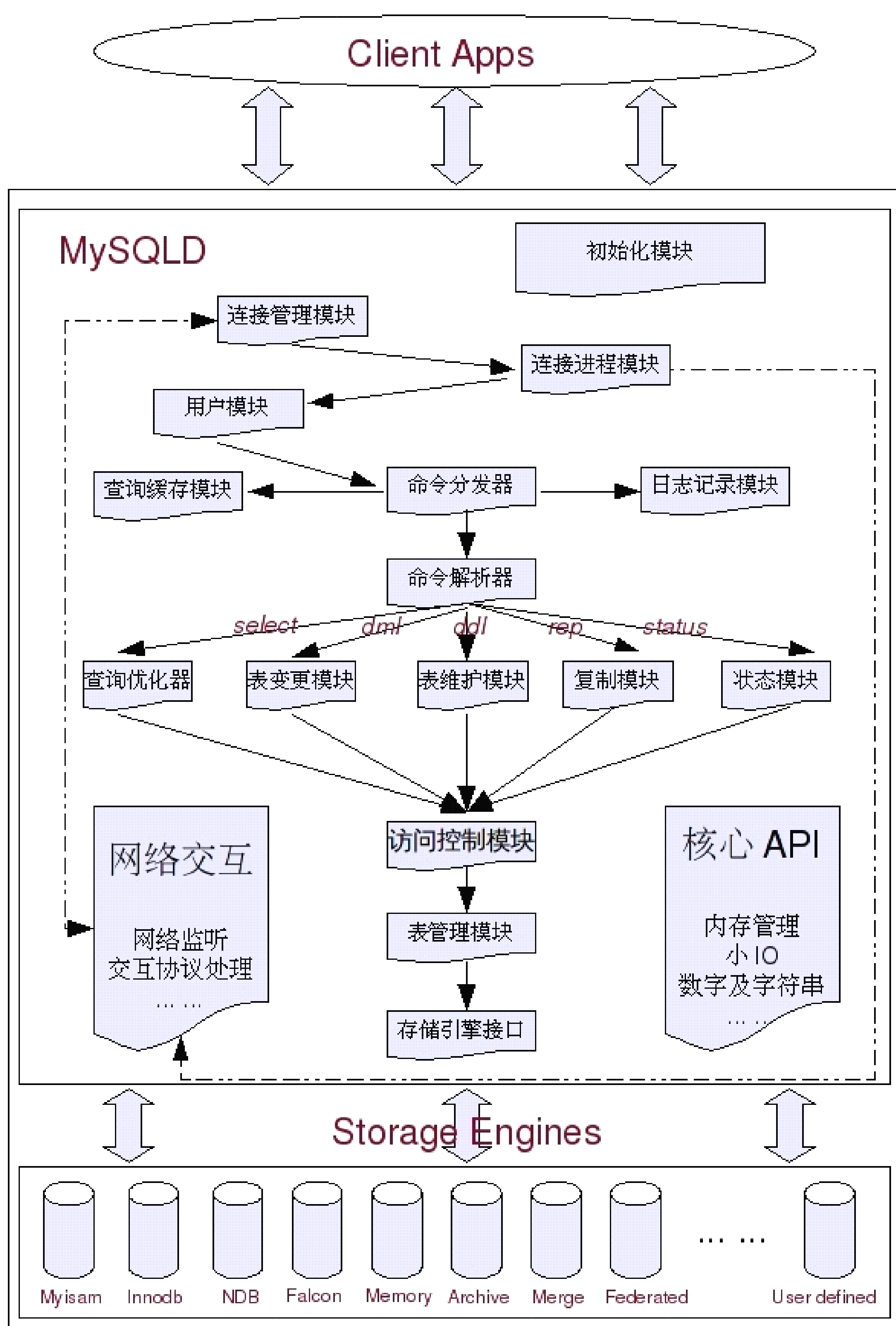

Client Apps
MySQLD
初始化模块
连接管理模块
连接进程模块
用户模块
查询缓存模块
命令分发器
日志记录模块
命令解析器
select
dml
ddl
rep
status
查询优化器
表变更模块
表维护模块
复制模块
状态模块
网络交互
网络监听
交互协议处理
… …
访问控制模块
表管理模块
存储引擎接口
核心 API
内存管理
小 IO
数字及字符串
… …
Storage Engines
Myisam
Innodb
NDB
Falcon
Memory
Archive
Merge
Federated
… …
User defined

图 2-2

2．3 MySQL 自带工具使用介绍

MySQL 数据库不仅提供了数据库的服务器端应用程序，同时还提供了大量的客户端工具程序，如 mysql，mysqladmin，mysqldump 等等，都是大家所熟悉的。虽然有些人对这些工具的功能都已经比较了解了，但是真正能将这些工具程序物尽其用的人可能并不是太多，或者知道的不全，也可能并不完全了解其中的某种特性。所以在这里我也简单地做一个介绍。

1、mysql

相信在所有 MySQL 客户端工具中，读者了解最多的就是 mysql 了，用的最多的应该也非他莫属。mysql 的功能和 Oracle 的 sqlplus 一样，为用户提供一个命令行接口来操作管理 MySQL 服务器。其基本的使用语法这里就不介绍了，大家只要运行一下“mysql --help”就会得到如下相应的基本使用帮助信息：

```
sky@sky:~$ mysql --help
mysql  Ver 14.14 Distrib 5.1.26-rc, for pc-linux-gnu (i686) using  EditLine
wrapper
Copyright (C) 2000-2008 MySQL AB
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL license
Usage: mysql [OPTIONS] [database]
  -?, --help          Display this help and exit.

  ... ...

  -e, --execute=name  Execute command and quit. (Disables --force and history
                      file)
  -E, --vertical      Print the output of a query (rows) vertically.

  ... ...

  -H, --html          Produce HTML output.
  -X, --xml           Produce XML output

  ... ...

  --prompt=name       Set the mysql prompt to this value.

  ... ...

  --tee=name          Append everything into outfile. See interactive help (\h)
                      also. Does not work in batch mode. Disable with
                      --disable-tee. This option is disabled by default.
```

```
... ...

-U, --safe-updates  Only allow UPDATE and DELETE that uses keys.
--select_limit=#    Automatic limit for SELECT when using --safe-updates
--max_join_size=#   Automatic limit for rows in a join when using
                    --safe-updates
... ...

--show-warnings     Show warnings after every statement.

... ...
```

上面的内容仅仅只是输出的一部分，省略去掉了大家最常用的一些参数（因为大家应该已经很熟悉了），留下了部分个人认为可能不是太经常用到，但是在有些情况下却能给我们带来意料之外的惊喜的一些参数选项。

首先看看“-e, --execute=name”参数，这个参数是告诉mysql，我只要执行“-e”后面的某个命令，而不是要通过mysql连接登录到MySQL Server上面。此参数在我们写一些基本的MySQL检查和监控的脚本中非常有用，我个人就经常在脚本中使用到他。

如果在连接时候使用了“-E, --vertical”参数，登入之后的所有查询结果都将以纵列显示，效果和我们在一条query之后以“\G”结尾一样，这个参数的使用场景可能不是特别多。

“-H, --html”与“-X, --xml”这两个参数很有意思的，在启用这两个参数之后，select出来的所有结果都会按照“Html”与“Xml”格式来输出，在有些场合之下，比如希望Xml或者Html文件格式导出某些报表文件的时候，是非常方便的。

“--prompt=name”参数对于做运维的人来说是一个非常重要的参数选项，其主要功能是定制自己的mysql提示符的显示内容。在默认情况下，我们通过mysql登入到数据库之后，mysql的提示符只是一个很简单的内容”mysql>“，没有其他任何附加信息。非常幸运的是mysql通过“--prompt=name”参数给我们提供了自定义提示信息的办法，可以通过配置显示登入的主机地址，登录用户名，当前时间，当前数据库schema，MySQL Server的一些信息等等。我个人强烈建议将登录主机名，登录用户名和所在的schema这三项加入提示内容，因为当大家手边管理的MySQL越来越多，操作越来越频繁的时候，非常容易因为操作的时候没有太在意自己当前所处的环境而造成在错误的环境执行了错误的命令并造成严重后果的情况。如果我们在提示内容中加入了这几项之后，至少可以更方便的提醒自己当前所处环境，以尽量减少犯错误的概率。

我个人的提示符定义：〞\\u@\\h : \\d \\r:\\m:\\s> 〞，显示效果：
“sky@localhost : test 04:25:45>”

“--tee=name”参数也是对运维人员非常有用的参数选项，用来告诉mysql，将所有输入和输出内容都记录进文件。在我们一些较大维护变更的时候，为了方便被查，最好是将整个操作过程的所有输入和输出内容都保存下来。有了“--tee=name”参数，就再也不用通过copy屏幕来保存操作过程了。

“-U, --safe-updates”,“--select_limit=#”和“--max_join_size=#”三个参数都是出于性能相关考虑的参数。使用“-U, --safe-updates”参数之后，将禁止所有不能使用索引的 update 和 delete 操作的请求，“--select_limit=#”的使用前提是有“-U, --safe-updates”参数，功能是限制查询记录的条数，“--max_join_size=#”也需要与“-U, --safe-updates”一起使用，限制参与 join 的最大记录数。

“--show-warnings”参数作用是在执行完每一条 query 之后都会自动执行一次“show warnings”，显示出最后一次 warning 的内容。

上面仅仅介绍了部分不是太常使用但是很有特点的少数几个参数选项，实际上 mysql 程序支持非常多的参数选项，有其自身的参数，也有提交给 MySQL Server 的。mysql 的所有参数选项都可以写在 MySQL Server 启动参数文件（my.cnf）的[mysql]参数 group 中，还有部分连接选项参数会从[client]参数 group 中读取，这样很多参数就可以不用在每次执行 mysql 的时候都手工输入，而由 mysql 程序自己自动从 my.cnf 文件 load 这些参数。

如果读者朋友希望对 mysql 其他参数选项或者 mysql 的其他更国有图有更深入的了解，可以通过 MySQL 官方参考手册查阅，也可以通过执行“mysql --help”得到帮助信息之后通过自行实验来做进一步的深刻认识。当然如果您是一位基本能看懂 c 语言的朋友，那么您完全可以通过 mysql 程序的源代码来发现其更多有趣的内容。

2、mysqladmin

```
Usage: mysqladmin [OPTIONS] command command ...
```

mysqadmin，顾名思义，提供的功能都是与 MySQL 管理相关的各种功能。如 MySQL Server 状态检查，各种统计信息的 flush，创建/删除数据库，关闭 MySQL Server 等等。mysqladmin 所能做的事情，虽然大部分都可以通过 mysql 连接登录上 MySQL Server 之后来完成，但是大部分通过 mysqladmin 来完成操作会更简单更方便。这里我将介绍一下自己经常使用到的几个常用功能：

ping 命令可以很容易检测 MySQL Server 是否还能正常提供服务

```
sky@sky:~# mysqladmin -u sky -ppwd -h localhost ping
mysqld is alive
```

status 命令可以获取当前 MySQL Server 的几个基本的状态值：

```
sky@sky:~# mysqladmin -u sky -ppwd -h localhost status
Uptime: 20960  Threads: 1  Questions: 75  Slow queries: 0  Opens: 15  Flush tables: 1  Open tables: 9  Queries per second avg: 0.3
```

processlist 获取当前数据库的连接线程信息：

```
sky@sky:~# mysqladmin -u sky -ppwd -h localhost processlist
+----+------+-----------+----+---------+------+-------+------------------+
| Id | User | Host      | db | Command | Time | State | Info             |
```

```
+----+------+-----------+----+---------+------+-------+------------------+
| 48 | sky  | localhost |    | Query   | 0    |       | show processlist |
+----+------+-----------+----+---------+------+-------+------------------+
```

上面的这三个功能是我在自己的一些简单监控脚本中经常使用到的，虽然得到的信息还是比较有限，但是对于完成一些比较基本的监控来说，已经足够胜任了。此外，还可以通过 mysqladmin 来 start slave 和 stop slave，kill 某个连接到 MySQL Server 的线程等等。

3、mysqldump

```
Usage: mysqldump [OPTIONS] database [tables]
OR     mysqldump [OPTIONS] --databases [OPTIONS] DB1 [DB2 DB3...]
OR     mysqldump [OPTIONS] --all-databases [OPTIONS]
```

mysqldump 这个工具我想大部分读者可能都比较熟悉了，其功能就是将 MySQL Server 中的数据以 SQL 语句的形式从数据库中 dump 成文本文件。虽然 mysqldump 是做为 MySQL 的一种逻辑备份工具为大家所认识，但我个人觉得称他为 SQL 生成导出工具更合适一点，因为通过 mysqldump 所生成的文件，全部是 SQL 语句，包括数据库和表的创建语句。当然，通过给 mysqldump 程序加"-T"选项参数之后，可以生成非 SQL 形式的指定给是的文本文件。这个功能实际上是调用了 MySQL 中的"select * into OUTFILE from ..."语句而实现。也可以通过"-d,--no-data"仅仅生成结构创建的语句。在声称 SQL 语句的时候，字符集设置这一项也是比较关键的，建议每次执行 mysqldump 程序的时候都通过尽量做到"--default-character-set=name"显式指定字符集内容，以防止以错误的字符集生成不可用的内容。mysqldump 所生成的 SQL 文件可以通过 mysql 工具执行。

4、mysqlimport

```
Usage: mysqlimport [OPTIONS] database textfile ...
```

mysqlimport 程序是一个将以特定格式存放的文本数据（如通过"select * into OUTFILE from ..."所生成的数据文件）导入到指定的 MySQL Server 中的工具程序，比如将一个标准的 csv 文件导入到某指定数据库的指定表中。mysqlimport 工具实际上也只是"load data infile"命令的一个包装实现。

5、mysqlbinlog

```
Usage: mysqlbinlog [OPTIONS] log-files
```

mysqlbinlog 程序的主要功能就是分析 MySQL Server 所产生的二进制日志（也就是大家所熟知的 binlog)。当我们希望通过之前备份的 binlog 做一些指定时间之类的恢复的时候，mysqlbinlog 就可以帮助我们找到恢复操作需要做哪些事情。通过 mysqlbinlog，我们可以解析出 binlog 中指定时间段或者指定日志起始和结束位置的内容解析成 SQL 语句，并导出到指定的文件中，在解析过程中，还可以通过指定数据库名称来过滤输出内容。

6、mysqlcheck

```
Usage: mysqlcheck [OPTIONS] database [tables]
```

OR　　mysqlcheck [OPTIONS] --databases DB1 [DB2 DB3...]
OR　　mysqlcheck [OPTIONS] --all-databases

mysqlcheck 工具程序可以检查（check），修复（repair），分析（analyze）和优化（optimize）MySQL Server 中的表，但并不是所有的存储引擎都支持这里所有的四个功能，像 Innodb 就不支持修复功能。实际上，mysqlcheck 程序的这四个功能都可以通过 mysql 连接登录到 MySQL Server 之后来执行相应命令完成完全相同的任务。

7、myisamchk
Usage: myisamchk [OPTIONS] tables[.MYI]

功能有点类似“mysqlcheck -c/-r”，对检查和修复 MyISAM 存储引擎的表，但只能对 MyISAM 存储引擎的索引文件有效，而且不用登录连接上 MySQL Server 即可完成操作。

8、myisampack
Usage: myisampack [OPTIONS] filename ...

对 MyISAM 表进行压缩处理，以缩减占用存储空间，一般主要用在归档备份的场景下，而且压缩后的 MyISAM 表会变成只读，不能进行任何修改操作。当我们希望归档备份某些历史数据表，而又希望该表能够提供较为高效的查询服务的时候，就可以通过 myisampack 工具程序来对该 MyISAM 表进行压缩，因为即使虽然更换成 archive 存储引擎也能够将表变成只读的压缩表，但是 archive 表是没有索引支持的，而通过压缩后的 MyISAM 表仍然可以使用其索引。

9、mysqlhotcopy
Usage: mysqlhotcopy db_name[./table_regex/] [new_db_name | directory]

mysqlhotcopy 和其他的客户端工具程序不太一样的是他不是 c（或者 c++）程序编写的，而是一个 perl 脚本程序，仅能在 Unix/Linux 环境下使用。他的主要功能就是对 MySQL 中的 MyISAM 存储引擎的表进行在线备份操作，其备份操作实际上就是通过对数据库中的表进行加锁，然后复制其结构，数据和索引文件来完成备份操作，当然，也可以通过指定“--noindices”告诉 mysqlhotcopy 不需要备份索引文件。

10、其他工具
除了上面介绍的这些工具程序之外，MySQL 还有自带了其他大量的工具程序，如针对离线 Innodb 文件做 checksum 的 innochecksum，转换 mSQL C API 函数的 msql2mysql，dumpMyISAM 全文索引的 myisam_ftdump，分析处理 slowlog 的 mysqldumpslow，查询 mysql 相关开发包位置和 include 文件位置的 mysql_config，向 MySQL AB 报告 bug 的 mysqlbug，测试套件 mysqltest 和 mysql_client_test，批量修改表存储引擎类型的 mysql_convert_table_format，能从更新日志中提取给定匹配规则的 query 语句的 mysql_find_rows，更改 MyIsam 存储引擎表后缀名的 mysql_fix_extensions，修复系统表的 mysql_fix_privilege_tables，查看数据库相关对象结构的 mysqlshow，MySQL 升级工具 mysql_upgrade，通过给定匹配模式来 kill 客户端连接线程的 mysql_zap，查看错误号信息的 perror，文本替换工具 replace，等等一系列工具程序可供我们使用。如果您希望在 MySQL

源代码的基础上做一些自己的修改，如修改 MyISAM 存储引擎的时候，可以利用 myisamlog 来进行跟踪分析 MyISAM 的 log。

2．4 小结

第 3 章 MySQL 存储引擎简介

前言

3．1 MySQL 存储引擎概述

MyISAM 存储引擎是 MySQL 默认的存储引擎，也是目前 MySQL 使用最为广泛的存储引擎之一。他的前身就是我们在 MySQL 发展历程中所提到的 ISAM，是 ISAM 的升级版本。在 MySQL 最开始发行的时候是 ISAM 存储引擎，而且实际上在最初的时候，MySQL 甚至是没有存储引擎这个概念的。MySQL 在架构上面也没有像现在这样的 sql layer 和 storage engine layer 这两个结构清晰的层次结构，当时不管是代码本身还是系统架构，对于开发者来说都很痛苦的一件事情。到后来，MySQL 意识到需要更改架构，将前 端的业务逻辑和后端数据存储以清晰的层次结构拆分开的同时，对 ISAM 做了功能上面的扩展和代码的重构，这就是 MyISAM 存储引擎的由来。

MySQL 在 5.1（不包括）之前的版本中，存储引擎是需要在 MySQL 安装的时候就必须和 MySQL 一起被编译并同时被安装的。也就是说，5.1 之前的 版本中，虽然存储引擎层和 sql 层的耦合已经非常少了，基本上完全是通过接口来实现交互，但是这两层之间仍然是没办法分离的，即使在安装的时候也是一样。

但是从 MySQL5.1 开始，MySQL AB 对其结构体系做了较大的改造，并引入了一个新的概念：插件式存储引擎体系结构。MySQL AB 在架构改造的时候，让存储引擎层和 sql 层各自更为独立，耦合更小，甚至可以做到在线加载信的存储引擎，也就是完全可以将一个新的存储引擎加载到一个正在 运行的 MySQL 中，而不影响 MySQL 的正常运行。插件式存储引擎的架构，为存储引擎的加载和移出更为灵活方便，也使自行开发存储引擎更为方便简单。在 这一点上面，目前还没有哪个数据库管理系统能够做到。

MySQL 的插件式存储引擎主要包括 MyISAM，Innodb，NDB Cluster，Maria，Falcon，Memory，Archive，Merge，Federated 等，其中最著名而且使用最为广泛的 MyISAM 和 Innodb 两种存储引擎。MyISAM 是 MySQL 最早的 ISAM 存储引擎的升级版本，也是 MySQL 默认的存储引擎。而 Innodb 实 际上并不是 MySQ 公司的，而是第三方软件公司 Innobase（在 2005 年被 Oracle 公司所收购）所开发，其最大的特点是提供了事务控制等特性， 所以使用者也非

常广泛。

其他的一些存储引擎相对来说使用场景要稍微少一些，都是应用于某些特定的场景，如 NDB Cluster 虽然也支持事务，但是主要是用于分布式环境，属于一个 share nothing 的分布式数据库存储引擎。Maria 是 MySQL 最新开发（还没有发布最终的 GA 版本）的对 MyISAM 的升级版存储引擎,Falcon 是 MySQL 公司自行研发的为了替代当前的 Innodb 存储引擎的一款带有事务等高级特性的数据库存储引擎，目前正在研发阶段。Memory 存储引擎所有数 据和索引均存储于内存中，所以主要是用于一些临时表，或者对性能要求极高，但是允许在西噢他嗯 Crash 的时候丢失数据的特定场景下。Archive 是一 个数据经过高比例压缩存放的存储引擎，主要用于存放过期而且很少访问的历史信息，不支持索引。Merge 和 Federated 在严格意义上来说，并不能算 作一个存储引擎。因为 Merge 存储引擎主要用于将几个基表 merge 到一起，对外作为一个表来提供服务，基表可以基于其他的几个存储引擎。而 Federated 实际上所做的事情，有点类似于 Oracle 的 dblink，主要用于远程存取其他 MySQL 服务器上面的数据。

3．2 MyISAM 存储引擎简介

MyISAM 存储引擎的表在数据库中，每一个表都被存放为三个以表名命名的物理文件。首先肯定会有任何存储引擎都不可缺少的存放表结构定义信息的.frm 文件，另外还有.MYD 和.MYI 文件，分别存放了表的数据（.MYD）和索引数据（.MYI）。每个表都有且仅有这样三个文件做为 MyISAM 存储类型的表的存储，也就是说不管这个表有多少个索引，都是存放在同一个.MYI 文件中。

MyISAM 支持以下三种类型的索引：

1、B-Tree 索引

B-Tree 索引，顾名思义，就是所有的索引节点都按照 balance tree 的数据结构来存储，所有的索引数据节点都在叶节点。

2、R-Tree 索引

R-Tree 索引的存储方式和 b-tree 索引有一些区别，主要设计用于为存储空间和多维数据的字段做索引，所以目前的 MySQL 版本来说，也仅支持 geometry 类型的字段作索引。

3、Full-text 索引

Full-text 索引就是我们长说的全文索引，他的存储结构也是 b-tree。主要是为了解决在我们需要用 like 查询的低效问题。

MyISAM 上面三种索引类型中，最经常使用的就是 B-Tree 索引了，偶尔会使用到 Full-text，但是 R-Tree 索引一般系统中都是很少用到的。另外 MyISAM 的 B-Tree 索引有一个较大的限制，那就是参与一个索引的所有字段的长度之和不能超过 1000 字节。

虽然每一个 MyISAM 的表都是存放在一个相同后缀名的.MYD 文件中，但是每个文件的存放格式实际上可能并不是完全一样的，因为 MyISAM 的数据存放格式是分为静态（FIXED）固

定长度、动态（DYNAMIC）可变长度以及压缩（COMPRESSED）这三种格式。当然三种格式中是否压缩是完全可以任由我们自己选择的，可以在创建表的时候通过 ROW_FORMAT 来指定{COMPRESSED | DEFAULT}，也可以通过 myisampack 工具来进行压缩，默认是不压缩的。而在非压缩的情况下，是静态还是动态，就和我们表中个字段的定义相关了。只要表中有可变长度类型的字段存在，那么该表就肯定是 DYNAMIC 格式的，如果没有任何可变长度的字段，则为 FIXED 格式，当然，你也可以通过 alter table 命令，强行将一个带有 VARCHAR 类型字段的 DYNAMIC 的表转换为 FIXED，但是所带来的结果是原 VARCHAR 字段类型会被自动转换成 CHAR 类型。相反如果将 FIXED 转换为 DYNAMIC，也会将 CHAR 类型字段转换为 VARCHAR 类型，所以大家手工强行转换的操作一定要谨慎。

MyISAM 存储引擎的表是否足够可靠呢？在 MySQL 用户参考手册中列出在遇到如下情况的时候可能会出现表文件损坏：

1、当 mysqld 正在做写操作的时候被 kill 掉或者其他情况造成异常终止；

2、主机 Crash；

3、磁盘硬件故障；

4、MyISAM 存储引擎中的 bug？

MyISAM 存储引擎的某个表文件出错之后，仅影响到该表，而不会影响到其他表，更不会影响到其他的数据库。如果我们的出据苦正在运行过程中发现某个 MyISAM 表出现问题了，则可以在线通过 check table 命令来尝试校验他，并可以通过 repair table 命令来尝试修复。在数据库关闭状态下，我们也可以通过 myisamchk 工具来对数据库中某个（或某些）表进行检测或者修复。不过强烈建议不到万不得已不要轻易对表进行修复操作，修复之前尽量做好可能的备份工作，以免带来不必要的后果。

另外 MyISAM 存储引擎的表理论上是可以被多个数据库实例同时使用同时操作的，但是不论是我们都不建议这样做，而且 MySQL 官方的用户手册中也有提到，建议尽量不要在多个 mysqld 之间共享 MyISAM 存储文件。

3．3 Innodb 存储引擎简介

在 MySQL 中使用最为广泛的除了 MyISAM 之外，就非 Innodb 莫属了。Innodb 做为第三方公司所开发的存储引擎，和 MySQL 遵守相同的开源 License 协议。

Innodb 之所以能如此受宠，主要是在于其功能方面的较多特点：

1、支持事务安装

Innodb 在功能方面最重要的一点就是对事务安全的支持，这无疑是让 Innodb 成为 MySQL 最为流行的存储引擎之一的一个非常重要原因。而且实现了 SQL92 标准所定义的所有四个级别（READ UNCOMMITTED，READ COMMITTED，REPEATABLE READ 和 SERIALIZABLE）。对事务安全的支持，无疑让很多之前因为特殊业务要求而不得不放弃使用 MySQL 的用户转向支持 MySQL，以及之前对数据库选型持观望态度的用户，也大大增加了对 MySQL 好感。

2、数据多版本读取

Innodb 在事务支持的同时，为了保证数据的一致性已经并发时候的性能，通过对 undo 信息，实现了数据的多版本读取。

3、锁定机制的改进

Innodb 改变了 MyISAM 的锁机制，实现了行锁。虽然 Innodb 的行锁机制的实现是通过索引来完成的，但毕竟在数据库中 99%的 SQL 语句都是要使用索引来做检索数据的。所以，行锁定机制也无疑为 Innodb 在承受高并发压力的环境下增强了不小的竞争力。

4、实现外键

Innodb 实现了外键引用这一数据库的重要特性，使在数据库端控制部分数据的完整性成为可能。虽然很多数据库系统调优专家都建议不要这样做，但是对于不少用户来说在数据库端加如外键控制可能仍然是成本最低的选择。

除了以上几个功能上面的亮点之外，Innodb 还有很多其他一些功能特色常常带给使用者不小的惊喜，同时也为 MySQL 带来了更多的客户。

在物理存储方卖弄，Innodb 存储引擎也和 MyISAM 不太一样，虽然也有.frm 文件来存放表结构定义相关的元数据，但是表数据和索引数据是存放在一起的。至于是每个表单独存放还是所有表存放在一起，完全由用户来决定（通过特定配置），同时还支持符号链接。

Innodb 的物理结构分为两大部分：

1、数据文件（表数据和索引数据）

存放数据表中的数据和所有的索引数据，包括主键和其他普通索引。在 Innodb 中，存在了表空间（tablespace）这样一个概念，但是他和 Oracle 的表空间又有较大的不同。首先，Innodb 的表空间分为两种形式。一种是共享表空间，也就是所有表和索引数据被存放在同一个表空间（一个或多个数据文件）中，通过 innodb_data_file_path 来指定，增加数据文件需要停机重启。 另外一种是独享表空间，也就是每个表的数据和索引被存放在一个单独的.ibd 文件中。

虽然我们可以自行设定使用共享表空间还是独享表空间来存放我们的表，但是共享表空间都是必须存在的，因为 Innodb 的 undo 信息和其他一些元数据信息都是存放在共享表空间里面的。共享表空间的数据文件是可以设置为固定大小和可自动扩展大小两种形式的，自动扩展形式的文件可以设置文件的最大大小和每次扩展量。在创建自动扩展的数据文件的时候，建议大家最好加上最大尺寸的属性，一个原因是文件系统本身是有一定大小限制的（但是 Innodb 并不知道），还有一个原因就是自身维护的方便。另外，Innodb 不仅可以使用文件系统，还可以使用原始块设备，也就是我们常说的裸设备。

当我们的文件表空间快要用完的时候，我们必须要为其增加数据文件，当然，只有共享表空间有此操作。共享表空间增加数据文件的操作比较简单，只需要在 innodb_data_file_path 参数后面按照标准格式设置好文件路径和相关属性即可，不过这里有一点需要注意的，就是 Innodb 在创建新数据文件的时候是不会创建目录的，如果指定目录不存在，则会报错并无法启动。另外一个较为令人头疼的就是 Innodb 在给共享表空间增加数据文件之后，必须要重启数据库系统才能生效，如果是使用裸设备，还需要有两次重启。这也是我一直不太喜欢使用共享表空间而选用独享表空间的原因之一。

2、日志文件

Innodb 的日志文件和 Oracle 的 redo 日志比较类似，同样可以设置多个日志组（最少 2 个），同样采用轮循策略来顺序的写入，甚至在老版本中还有和 Oracle 一样的日志归档特性。如果你的数据库中有创建了 Innodb 的表，那么千万别全部删除 innodb 的日志文件，因为很可能就会让你的数据库 crash，无法启动，或者是丢失数据。

由于 Innodb 是事务安全的存储引擎，所以系统 Crash 对他来说并不能造成非常严重的损失，由于有 redo 日志的存在，有 checkpoint 机制的保护，Innodb 完全可以通过 redo 日志将数据库 Crash 时刻已经完成但还没有来得及将数据写入磁盘的事务恢复，也能够将所有部分完成并已经写入磁盘的未完成事务回滚并将数据还原。

Innodb 不仅在功能特性方面和 MyISAM 存储引擎有较大区别，在配置上面也是单独处理的。在 MySQL 启动参数文件设置中，Innodb 的所有参数基本上都带有前缀"innodb_"，不论是 innodb 数据和日志相关，还是其他一些性能，事务等等相关的参数都是一样。和所有 Innodb 相关的系统变量一样，所有的 Innodb 相关的系统状态值也同样全部以"Innodb_"前缀。当然，我们也完全可以仅仅通过一个参数（skip-innodb）来屏蔽 MySQL 中的 Innodb 存储引擎，这样即使我们在安装编译的时候将 Innodb 存储引擎安装进去了，使用者也无法创建 Innodb 的表。

3．4 NDB Cluster 存储引擎简介

NDB 存储引擎也叫 NDB Cluster 存储引擎，主要用于 MySQL Cluster 分布式集群环境，Cluster 是 MySQL 从 5.0 版本才开始提供的新功能。这部分我们可能并不仅仅只是介绍 NDB 存储引擎，因为离开了 MySQL CLuster 整个环境，NDB 存储引擎也将失去太多意义。 所以这一节主要是介绍一下 MySQL Cluster 的相关内容。

简单的说，Mysql Cluster 实际上就是在无共享存储设备的情况下实现的一种内存数据库 Cluster 环境，其主要是通过 NDB Cluster（简称 NDB）存储引擎来实现的。

一般来说，一个 Mysql Cluster 的环境主要由以下三部分组成：

a) 负责管理各个节点的 Manage 节点主机：

管理节点负责整个 Cluster 集群中各个节点的管理工作，包括集群的配置，启动关闭各节点，以及实施数据的备份恢复等。管理节点会获取整个 Cluster 环境中各节点的状态和错误信息，并且将各 Cluster 集群中各个节点的信息反馈给整个集群中其他的所有节点。由于管理节点上保存在整个 Cluster 环境的配置，同时担任了集群中各节点的基本沟通工作，所以他必须是最先被启动的节点。

b) SQL 层的 SQL 服务器节点（后面简称为 SQL 节点），也就是我们常说的 Mysql Server:

主要负责实现一个数据库在存储层之上的所有事情，比如连接管理，query 优化和响应，cache 管理等等，只有存储层的工作交给了 NDB 数据节点去处理了。也就是说，在纯粹的 Mysql Cluster 环境中的 SQL 节点，可以被认为是一个不需要提供任何存储引擎的 Mysql 服务器，因为他的存储引擎有 Cluster 环境中的 NDB 节点来担任。所以，SQL 层各 Mysql 服

务器的启动与普通的 Mysql 启动有一定的区别，必须要添加 ndbcluster 项，可以添加在 my.cnf 配置文件中，也可以通过启动命令行来指定。

c）Storage 层的 NDB 数据节点，也就是上面说的 NDB Cluster：

NDB 是一个内存式存储引擎也就是说，他会将所有的数据和索引数据都 load 到内存中，但也会将数据持久化到存储设备上。不过，最新版本，已经支持用户自己选择数据可以不全部 Load 到内存中了，这对于有些数据量太大或者基于成本考虑而没有足够内存空间来存放所有数据的用户来说的确是一个大好消息。

NDB 节点主要是实现底层数据存储的功能，保存 Cluster 的数据。每一个 NDB 节点保存完整数据的一部分（或者一份完整的数据，视节点数目和配置而定），在 MySQL CLuster 里面叫做一个 fragment。而每一个 fragment，正常情况来讲都会在其他的主机上面有一份（或者多分）完全相同的镜像存在。这些都是通过配置来完成的，所以只要配置得当，Mysql Cluster 在存储层不会出现单点的问题。一般来说，NDB 节点被组织成一个一个的 NDB Group，一个 NDB Group 实际上就是一组存有完全相同的物理数据的 NDB 节点群。

上面提到了 NDB 各个节点对数据的组织，可能每个节点都存有全部的数据也可能只保存一部分数据，主要是受节点数目和参数来控制的。首先在 Mysql Cluster 主配置文件（在管理节点上面，一般为 config.ini）中，有一个非常重要的参数叫 NoOfReplicas，这个参数指定了每一份数据被冗余存储在不同节点上面的份数，该参数一般至少应该被设置成 2，也只需要设置成 2 就可以了。因为正常来说，两个互为冗余的节点同时出现故障的概率还是非常小的，当然如果机器和内存足够多的话，也可以继续增大。一个节点上面是保存所有的数据还是一部分数据，还受到存储节点数目的限制。NDB 存储引擎首先保证 NoOfReplicas 参数配置的要求对数据冗余，来使用存储节点，然后再根据节点数目将数据分段来继续使用多余的 NDB 节点，分段的数目为节点总数除以 NoOfReplicas 所得。

MySQL Cluster 本身所包含的内容非常之多，出于篇幅考虑，这里暂时不做很深入的介绍，在本书的架构设计部分的高可用性设计一章中将会有更为详细的介绍与实施细节，大家也可以通过 MySQL 官方文档来进一步了解部分细节。

3．5 其他存储引擎介绍

3.5.1 Merge 存储引擎：

MERGE 存储引擎，在 MySQL 用户手册中也提到了，也被大家认识为 MRG_MyISAM 引擎。Why？因为 MERGE 存储引擎可以简单的理解为其功能就是实现了对结构相同的 MyISAM 表，通过一些特殊的包装对外提供一个单一的访问入口，以达到减小应用的复杂度的目的。要创建 MERGE 表，不仅仅基表的结构要完全一致，包括字段的顺序，基表的索引也必须完全一致。

MERGE 表本身并不存储数据，仅仅只是为多个基表提供一个同意的存储入口。所以在创建 MERGE 表的时候，MySQL 只会生成两个较小的文件，一个是.frm 的结构定义文件，还有一个.MRG 文件，用于存放参与 MERGE 的表的名称（包括所属数据库 schema）。之所以需要有所

属数据库的 schema，是因为 MERGE 表不仅可以实现将 Merge 同一个数据库中的表，还可以 Merge 不同数据库中的表，只要是权限允许，并且在同一个 mysqld 下面，就可以进行 Merge。MERGE 表在被创建之后，仍然可以通过相关命令来更改底层的基表。

MERGE 表不仅可以提供读取服务，也可以提供写入服务。要让 MERGE 表提供可 INSERT 服务，必须在在表被创建的时候就指明 INSERT 数据要被写入哪一个基表，可以通过 insert_method 参数来控制。如果没有指定该参数，任何尝试往 MERGE 表中 INSERT 数据的操作，都会出错。此外，无法通过 MERGE 表直接使用基表上面的全文索引，要使用全文索引，必须通过基表本身的存取才能实现。

3.5.2 Memory 存储引擎：

Memory 存储引擎，通过名字就很容易让人知道，他是一个将数据存储在内存中的存储引擎。Memory 存储引擎不会将任何数据存放到磁盘上，仅仅存放了一个表结构相关信息的.frm 文件在磁盘上面。所以一旦 MySQL Crash 或者主机 Crash 之后，Memory 的表就只剩下一个结构了。Memory 表支持索引，并且同时支持 Hash 和 B－Tree 两种格式的索引。由于是存放在内存中，所以 Memory 都是按照定长的空间来存储数据的，而且不支持 BLOB 和 TEXT 类型的字段。Memory 存储引擎实现页级锁定。

既然所有数据都存放在内存中，那么他对内存的消耗量是可想而知的。在 MySQL 的用户手册上面有这样一个公式来计算 Memory 表实际需要消耗的内存大小：

```
SUM_OVER_ALL_BTREE_KEYS(max_length_of_key + sizeof(char*) * 4)
            + SUM_OVER_ALL_HASH_KEYS(sizeof(char*) * 2)
            + ALIGN(length_of_row+1, sizeof(char*))
```

3.5.3 BDB 存储引擎：

BDB 存储引擎全称为 BerkeleyDB 存储引擎，和 Innodb 一样，也不是 MySQL 自己开发实现的一个存储引擎，而是由 Sleepycat Software 所提供，当然，也是开源存储引擎，同样支持事务安全。

BDB 存储引擎的数据存放也是每个表两个物理文件，一个.frm 和一个.db 的文件，数据和索引信息都是存放在.db 文件中。此外，BDB 为了实现事务安全，也有自己的 redo 日志，和 Innodb 一样，也可以通过参数指定日志文件存放的位置。在锁定机制方面，BDB 和 Memory 存储引擎一样，实现页级锁定。

由于 BDB 存储引擎实现了事务安全，那么他肯定也需要有自己的 check point 机制。BDB 在每次启动的时候，都会做一次 check point，并且将之前的所有 redo 日志清空。在运行过程中，我们也可以通过执行 flush logs 来手工对 BDB 进行 check point 操作。

3.5.4 FEDERATED 存储引擎：

FEDERATED 存储引擎所实现的功能，和 Oracle 的 DBLINK 基本相似，主要用来提供对远程 MySQL 服务器上面的数据的访问借口。如果我们使用源码编译来安装 MySQL，那么必须手工指定启用
FEDERATED 存储引擎才行，因为 MySQL 默认是不起用该存储引擎的。

当我们创建一个 FEDERATED 表的时候，仅仅在本地创建了一个表的结构定义信息的文件而已，所有数据均实时取自远程的 MySQL 服务器上面的数据库。

当我们通过 SQL 操作 FEDERATED 表的时候，实现过程基本如下：

a、SQL 调用被本地发布

b、MySQL 处理器 API（数据以处理器格式）

c、MySQL 客户端 API（数据被转换成 SQL 调用）

d、远程数据库-> MySQL 客户端 API

e、转换结果包（如果有的话）到处理器格式

f、处理器 API -> 结果行或受行影响的对本地的计数

3.5.5 ARCHIVE 存储引擎：

ARCHIVE 存储引擎主要用于通过较小的存储空间来存放过期的很少访问的历史数据。ARCHIVE 表不支持索引，通过一个.frm 的结构定义文件，一个.ARZ 的数据压缩文件还有一个.ARM 的 meta 信息文件。由于其所存放的数据的特殊性，ARCHIVE 表不支持删除，修改操作，仅支持插入和查询操作。锁定机制为行级锁定。

3.5.6 BLACKHOLE 存储引擎：

BLACKHOLE 存储引擎是一个非常有意思的存储引擎，功能恰如其名，就是一个“黑洞"。就像我们 unix 系统下面的“/dev/null”设备一样，不管我们写入任何信息，都是有去无回。那么 BLACKHOLE 存储引擎对我们有什么用呢？在我最初接触 MySQL 的时候我也有过同样的疑问，不知道 MySQL 提供这样一个存储引擎给我们的用意为何？但是后来在又一次数据的迁移过程中，正是 BLACKHOLE 给我带来了非常大的功效。在那次数据迁移过程中，由于数据需要经过一个中转的 MySQL 服务器做一些相关的转换操作，然后再通过复制移植到新的服务器上面。可当时我没有足够的空间来支持这个中转服务器的运作。这时候就显示出 BLACKHOLE 的功效了，他不会记录下任何数据，但是会在 binlog 中记录下所有的 sql。而这些 sql 最终都是会被复制所利用，并实施到最终的 slave 端。

MySQL 的用户手册上面还介绍了 BLACKHOLE 存储引擎其他几个用途如下：

a、SQL 文件语法的验证。

b、来自二进制日志记录的开销测量，通过比较允许二进制日志功能的 BLACKHOLE 的性

能与禁止二进制日志功能的 BLACKHOLE 的性能。

c、因为 BLACKHOLE 本质上是一个“no-op” 存储引擎，它可能被用来查找与存储引擎自身不相关的性能瓶颈。

3.5.7 CSV 存储引擎：

CSV 存储引擎实际上操作的就是一个标准的 CSV 文件，他不支持索引。起主要用途就是大家有些时候可能会需要通过数据库中的数据导出成一份报表文件，而 CSV 文件是很多软件都支持的一种较为标准的格式，所以我们可以通过先在数据库中建立一张 CVS 表，然后将生成的报表信息插入到该表，即可得到一份 CSV 报表文件了。

3．6 小结

多存储引擎是 MySQL 有别于其他数据库管理软件的最大特色，不同的存储引擎有不同的特点，可以应对不同的应用场景，这让我们在实际的应用中可以根据不同的应用特点来选择最有利的存储引擎，给了我们足够的灵活性。通过这一章对 MySQL 各个存储引擎的初步了解，我想各位读者朋友应该已经对 MySQL 的主要存储引擎有了一定的认识，在后续的章节中对于一些常用的存储引擎还会有更为深入的介绍。

第 4 章 MySQL 安全管理

前言

对于任何一个企业来说，其数据库系统中所保存数据的安全性无疑是非常重要的，尤其是公司的有些商业数据，可能数据就是公司的根本，失去了数据的安全性，可能就是失去了公司的一切。本章将针对 MySQL 的安全相关内容进行较为详细的介绍。

4．1 数据库系统安全相关因素

一、外围网络：

MySQL 的大部分应用场景都是基于网络环境的，而网络本身是一个充满各种入侵危险的环境，所以要保护他的安全，在条件允许的情况下，就应该从最外围的网络环境开始“布防"，因为这一层防线可以从最大范围内阻止可能存在的威胁。

在网络环境中，任意两点之间都可能存在无穷无尽的“道路”可以抵达，是一个真正“条条道路通罗马”的环境。在那许许多多的道路中，只要有一条道路不够安全，就可能被入侵者利用。当然，由于所处的环境不同，潜在威胁的来源也会不一样。有些 MySQL 所处环境是暴露在整个广域网中，可以说是完全“裸露”在任何可以接入网络环境的潜在威胁者面前。而有些 MySQL 是在一个环境相对小一些的局域网之内，相对来说，潜在威胁者也会少很多。处在局域网之内的 MySQL，由于有局域网出入口的网络设备的基本保护，相对于暴露在广域网中要安全不少，主要威胁对象基本上控制在了可以接入局域网的内部潜在威胁者，和极少数能够突破最外围防线（局域网出入口的安全设备）的入侵者。所以，尽可能的让我们的 MySQL 处在一个有保护的局域网之中，是非常必要的。

二、主机：

有了网络设备的保护，我们的 MySQL 就足够安全了么？我想大家都会给出否定的回答。因为即使我们局域网出入口的安全设备足够的强大，可以拦截住外围试图入侵的所有威胁者，但如果威胁来自局域网内部呢？比如局域网中可能存在被控制的设备，某些被控制的有权限接入局域网的设备，以及内部入侵者等都仍然是威胁者。所以说，吉使在第一层防线之内，我们仍然存在安全风险，局域网内部仍然会有不少的潜在威胁存在。

这个时候就需要我们部署第二道防线“主机层防线”了。“主机层防线”主要拦截网络（包括局域网内）或者直连的未授权用户试图入侵主机的行为。因为一个恶意入侵者在登录到主机之后，可能通过某些软件程序窃取到那些自身安全设置不够健壮的数据库系统的登入口令，从而达到窃取或者破坏数据的目的。如一个主机用户可以通过一个未删除且未设置密码的无用户名本地帐户轻易登入数据库，也可以通过 MySQL 初始安装好之后就存在的无密码的“root@localhost”用户登录数据库并获得数据库最高控制权限。

非法用户除了通过登入数据库获取（或者破坏）数据之外，还可能通过主机上面相关权限设置的漏洞，跳过数据库而直接获取 MySQL 数据（或者日志）文件达到窃取数据的目的，或者直接删除数据（或者日志）文件达到破坏数据的目的。

三、数据库：

通过第二道防线“主机层防线”的把守，我们又可以挡住很大一部分安全威胁者。但仍然可能有极少数突破防线的入侵者。而且即使没有任何“漏网之鱼"，那些有主机登入权限的使用者呢？是否真的就是完全可信任对象？No，我们不能轻易冒这个潜在风险。对于一个有足够安全意识的管理员来说，是不会轻易放任任何一个潜在风险存在的。

这个时候，我们的第三道防线，“数据库防线”就需要发挥他的作用了。“数据库防线"也就是 MySQL 数据库系统自身的访问控制授权管理相关模块。这道防线基本上可以说是 MySQL 的最后一道防线了，也是最核心最重要的防线。他首先需要能够抵挡住在之前的两层防线都没有能够阻拦住的所有入侵威胁，同时还要能够限制住拥有之前二层防线自由出入但不具备数据库访问权限的潜在威胁者，以确保数据库自身的安全以及所保存数据的安全。

之前的二层防线对于所有数据库系统来说基本上区别不大，都存在着基本相同的各种威胁，不论是 Oracle 还是 MySQL，以及任何其他的数据库管理系统，都需要基本一致的“布防”策略。但是这第三层防线，也就是各自自身的“数据库防线”对于每个数据库系统来说都存在较大的差异，因为每种数据库都有各自不太一样的专门负责访问授权相关功能的模块。不论是权限划分还是实现方式都可能不太一样。

对于 MySQL 来说，其访问授权相关模块主要是由两部分组成。一个是基本的用户管理模块，另一个是访问授权控制模块。用户管理模块的功能相对简单一些，主要是负责用户登录连接相关的基本权限控制，但其在安全控制方面的作用却不比任何环节小。他就像 MySQL 的一个“大门门卫”一样，通过校验每一位敲门者所给的进门“暗号"（登入口令），决定是否给敲门者开门。而访问授权控制模块则是随时随地检查已经进门的访问者，校验他们是否有访问所发出请求需要访问的数据的权限。通过校验者可顺利拿到数据，而未通过校验的访问者，只能收到“访问越权了”的相关反馈。

上面的三道防线组成了如图 4-1 所示的三道坚固的安全保护壁垒，就像三道坚固的城墙一样保护这 MySQL 数据库中的数据。只要保障足够，基本很难有人能够攻破这三道防线。

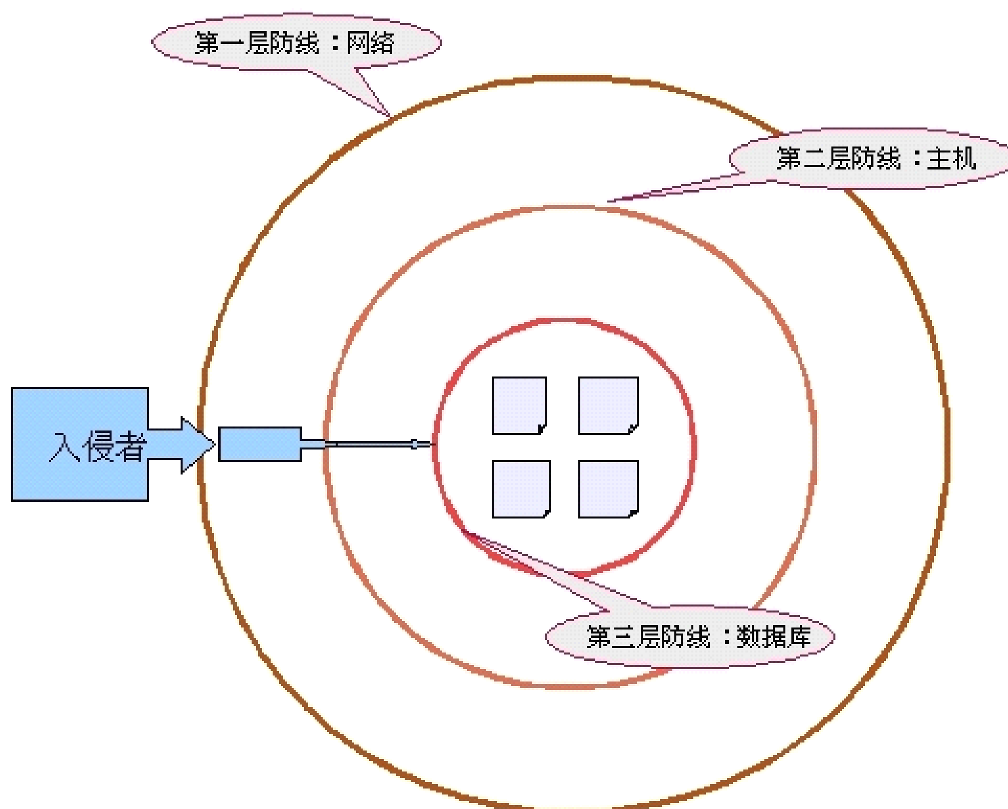


图 4-1

四、代码：

1、SQL 语句相关安全因素：

“SQL 注入攻击”这个术语我想大部分读者朋友都听说过了？指的就是攻击者根据数据库的 SQL 语句解析器的原理，利用程序中对客户端所提交数据的校验漏洞，从而通过程序动态提交数据接口提交非法数据，达到攻击者的入侵目的。

“SQL 注入攻击”的破坏性非常的大，轻者造成数据被窃取，重者数据遭到破坏，甚至可能丢失全部的数据。如果读者朋友还不是太清楚何为“SQL 注入攻击"，建议通过互联网搜索一下，可以得到非常多非常详细的介绍及案例分析，这里有不做详细介绍了。

2、程序代码相关安全因素：

程序代码如果权限校验不够仔细而存在安全漏洞，则同样可能会被入侵者利用，达到窃取数据等目的。比如，一个存在安全漏洞的信息管理系统，很容易就可能窃取到其他一些系统的登入口令。之后，就能堂而皇之的轻松登录相关系统达到窃取相关数据的目的。甚至还可能通过应用系统中保存不善的数据库系统连接登录口令，从而带来更大的损失。

4．2 MySQL 权限系统介绍

4.2.1 权限系统简介

MySQL 的权限系统在实现上比较简单，相关权限信息主要存储在几个被称为 grant tables 的系统表中，即：mysql.User，mysql.db，mysql.Host，mysql.table_priv 和 mysql.column_priv。由于权限信息数据量比较小，而且访问又非常频繁，所以 Mysql 在启动的时候，就会将所有的权限信息都 Load 到内存中保存在几个特定的结构中。所以才有我们每次手工修改了权限相关的表之后，都需要执行“FLUSH PRIVILEGES”命令重新加载 MySQL 的权限信息。当然，如果我们通过 GRANT，REVOKE 或者 DROP USER 命令来修改相关权限，则不需要手工执行 FLUSH PRIVILEGES 命令，因为通过 GRANT，REVOKE 或者 DROP USER 命令所做的权限修改在修改系统表的同时也会更新内存结构中的权限信息。在 MySQL5.0.2 或更高版本的时候，MySQL 还增加了 CREATE USER 命令，以此创建无任何特别权限（仅拥有初始 USAGE 权限）的用户，通过 CREATE USER 命令创建新了新用户之后，新用户的信息也会自动更新到内存结构中。所以，建议读者一般情况下尽量使用 GRANT，REVOKE，CREATE USER 以及 DROP USER 命令来进行用户和权限的变更操作，尽量减少直接修改 grant tables 来实现用户和权限变更的操作。

4.2.2 权限授予与去除

要为某个用户授权，可以使用 GRANT 命令，要去除某个用户已有的权限则使用 REVOKE 命令。当然，出了这两者之外还有一种比较暴力的办法，那就是直接更新 grant tables 系统表。当给某个用户授权的时候，不仅需要指定用户名，同时还要指定来访主机。如果在授权的时候仅指定用户名，则 MySQL 会自动认为是对'username'@'%'授权。要去除某个用户的的权限同样也需要指定来访主机。

可能有些时候我们还会需要查看某个用户目前拥有的权限，这可以通过两个方式实现，首先是通过执行“SHOW GRANTS FOR 'username'@'hostname'” 命令来获取之前该用户身上的所有授权。另一种方法是查询 grant tables 里面的权限信息。

4.2.3 权限级别

MySQL 中的权限分为五个级别，分别如下：

1、Global Level：

Global Level 的权限控制又称为全局权限控制，所有权限信息都保存在 mysql.user 表中。Global Level 的所有权限都是针对整个 mysqld 的，对所有的数据库下的所有表及所有字段都有效。如果一个权限是以 Global Level 来授予的，则会覆盖其他所有级别的相同权限设置。比如我们首先给 abc 用户授权可以 UPDATE 指定数据库如 test 的 t 表，然后又在全局级别 REVOKE 掉了 abc 用户对所有数据库的所有表的 UPDATE 权限。则这时候的 abc 用户将不再拥有用对 test.t 表的更新权限。Global Level 主要有如下这些权限(见表 4-1)：

表 4-1

名称	版本支持	限制信息
ALTER	ALL	表结构更改权限
ALTER ROUTINE	5.0.3+	procedure，function 和 trigger 等的变更权限
CREATE	ALL	数据库，表和索引的创建权限
CREATE ROUTINE	5.0.3+	procedure，function 和 trigger 等的变更权限
CREATE TEMPORARY TABLES	4.0.2+	临时表的创建权限
CREATE USER	5.0.3+	创建用户的权限
CREATE VIEW	5.0.1+	创建视图的权限
DELETE	All	删除表数据的权限
DROP	All	删除数据库对象的权限
EXECUTE	5.0.3+	procedure，function 和 trigger 等的执行权限
FILE	All	执行 LOAD DATA INFILE 和 SELECT ... INTO FILE 的权限
INDEX	All	在已有表上创建索引的权限
INSERT	All	数据插入权限
LOCK TABLES	4.0.2+	执行 LOCK TABLES 命令显示给表加锁的权限
PROCESS	All	执行 SHOW PROCESSLIST 命令的权限
RELOAD	All	执行FLUSH等让数据库重新Load某些对象或者数据的命令的权限
REPLICATION CLIENT	4.0.2+	执行 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令的权限
REPLICATION SLAVE	4.0.2+	复制环境中 Slave 连接用户所需要的复制权限
SELECT	All	数据查询权限
SHOW DATABASES	4.0.2+	执行 SHOW DATABASES 命令的权限
SHOW VIEW	5.0.1+	执行 SHOW CREATE VIEW 命令查看 view 创建语句的权限
SHUTDOWN	All	MySQL Server 的 shut down 权限（如通过 mysqladmin 执行 shutdown 命令所使用的连接用户）
SUPER	4.0.2+	执行 kill 线程，CHANGE MASTER, PURGE MASTER LOGS, and SET GLOBAL 等命令的权限
UPDATE	All	更新数据的权限
USAGE	All	新创建用户后不授任何权限的时候所拥有的最小权限

要授予 Global Level 的权限，则只需要在执行 GRANT 命令的时候，用“*.*”来指定适用范围是 Global 的即可，当有多个权限需要授予的时候，也并不需要多次重复执行 GRANT 命令，只需要一次将所有需要的权限名称通过逗号（“,”）分隔开即可，如下：

```
root@localhost : mysql 05:14:35> GRANT SELECT,UPDATE,DELETE,INSERT ON *.* TO 'def'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

2、Database Level

Database Level 是在 Global Level 之下，其他三个 Level 之上的权限级别，其作用域即为所指定整个数据库中的所有对象。与 Global Level 的权限相比，Database Level 主要少了以下几个权限:CREATE USER,FILE,PROCESS,RELOAD,REPLICATION CLIENT,REPLICATION SLAVE，SHOW DATABASES，SHUTDOWN，SUPER 和 USAGE 这几个权限，没有增加任何权限。之前我们说过 Global Level 的权限会覆盖底下其他四层的相同权限,Database Level 也一样，虽然他自己可能会被 Global Level 的权限设置所覆盖，但同时他也能覆盖比他更下层的 Table，Column 和 Routine 这三层的权限。

如果要授予 Database Level 的权限，则可以有两种实现方式：

1、在执行 GRANT 命令的时候，通过“database.*”来限定权限作用域为 database 整个数据库，如下：

```
root@localhost : mysql 06:06:26> GRANT ALTER ON test.* TO 'def'@'localhost';
Query OK, 0 rows affected (0.00 sec)

root@localhost : test 06:12:45> SHOW GRANTS FOR def@localhost;
+------------------------------------------------------------------+
| Grants for def@localhost                                         |
+------------------------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'def'@'localhost' |
| GRANT ALTER ON `test`.* TO 'def'@'localhost'                     |
+------------------------------------------------------------------+
```

2、先通过 USE 命令选定需要授权的数据库，然后通过“*”来限定作用域，这样授权的作用域实际上就是当前选定的整个数据库。

```
root@localhost : mysql 06:14:05> USE test;
Database changed
root@localhost : test 06:13:10> GRANT DROP ON * TO 'def'@'localhost';
Query OK, 0 rows affected (0.00 sec)

root@localhost : test 06:15:26> SHOW GRANTS FOR def@localhost;
+------------------------------------------------------------------+
| Grants for def@localhost                                         |
+------------------------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'def'@'localhost' |
| GRANT DROP, ALTER ON `test`.* TO 'def'@'localhost'               |
```

```
+------------------------------------------------------------------+
```

在授予权限的时候，如果有相同的权限需要授予多个用户，我们也可以在授权语句中一次写上多个用户信息，通过逗号（,）分隔开就可以了，如下：

```
root@localhost : mysql 05:22:32> grant create on perf.* to 'abc'@'localhost','def'@'localhost';
Query OK, 0 rows affected (0.00 sec)

root@localhost : mysql 05:22:46> SHOW GRANTS FOR def@localhost;
+------------------------------------------------------------------+
| Grants for def@localhost                                         |
+------------------------------------------------------------------+
| GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'def'@'localhost' |
| GRANT DROP, ALTER ON `test`.* TO 'def'@'localhost'               |
| GRANT CREATE ON `perf`.* TO 'def'@'localhost'                    |
        +------------------------------------------------------------------
+
3 rows in set (0.00 sec)

root@localhost : mysql 05:23:13> SHOW GRANTS FOR abc@localhost;
+------------------------------------------------------------------+
| Grants for abc@localhost                                         |
+------------------------------------------------------------------+
| GRANT CREATE ON `perf`.* TO 'abc'@'localhost'                    |
| GRANT SELECT ON `test`.* TO 'abc'@'localhost'                    |
+------------------------------------------------------------------+
3 rows in set (0.00 sec)
```

3、Table Level

Database Level 之下就是 Table Level 的权限了，Table Level 的权限可以被 Global Level 和 Database Level 的权限所覆盖，同时也能覆盖 Column Level 和 Routine Level 的权限。

Table Level 的权限作用范围是授权语句中所指定数据库的指定表。如可以通过如下语句给 test 数据库的 t1 表授权：

```
root@localhost : test 12:02:15> GRANT INDEX ON test.t1 TO 'abc'@'%.jianzhaoyang.com';
Query OK, 0 rows affected, 1 warning (0.00 sec)

root@localhost : test 12:02:53> SHOW GRANTS FOR 'abc'@'%.jianzhaoyang.com';
+----------------------------------------------------------+
| Grants for abc@*.jianzhaoyang.com                        |
+----------------------------------------------------------+
| GRANT USAGE ON *.* TO 'abc'@'%.jianzhaoyang.com'         |
```

```
| GRANT INDEX ON `test`.`t1` TO 'abc'@'%.jianzhaoyang.com' |
+----------------------------------------------------------+
```

上面的授权语句在测试给 test 数据库的 t1 表授予 Table Level 的权限的同时，还测试了将权限授予含有通配符“%”的所有“.jianzhaoyang.com”主机。其中的 USAGE 权限是每个用户都有的最基本权限。

Table Level 的权限由于其作用域仅限于某个特定的表，所以权限种类也比较少，仅有 ALTER，CREATE，DELETE，DROP，INDEX，INSERT，SELECT UPDATE 这八种权限。

4、Column Level

Column Level 的权限作用范围就更小了，仅仅是某个表的指定的某个（活某些）列。由于权限的覆盖原则，Column Level 的权限同样可以被 Global，Database，Table 这三个级别的权限中的相同级别所覆盖，而且由于 Column Level 所针对的权限和 Routine Level 的权限作用域没有重合部分，所以不会有覆盖与被覆盖的关系。针对 Column Level 级别的权限仅有 INSERT,SELECT 和 UPDATE 这三种。Column Level 的权限授权语句语法基本和 Table Level 差不多，只是需要在权限名称后面将需要授权的列名列表通过括号括起来，如下：

```
root@localhost : test 12:14:46> GRANT SELECT(id,value) ON test.t2 TO 'abc'@'%.jianzhaoyang.com';
Query OK, 0 rows affected(0.00 sec)

root@localhost : test 12:16:49> SHOW GRANTS FOR 'abc'@'%.jianzhaoyang.com';
+---------------------------------------------------------------------------+
| Grants for abc@*.jianzhaoyang.com                                         |
+---------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'abc'@'%.jianzhaoyang.com'                          |
| GRANT SELECT (value, id) ON `test`.`t2` TO 'abc'@'%.jianzhaoyang.com'     |
| GRANT INDEX ON `test`.`t1` TO 'abc'@'%.jianzhaoyang.com'                  |
+---------------------------------------------------------------------------+
```

注意：当某个用户在向某个表插入（INSERT）数据的时候，如果该用户在该表中某列上面没有 INSERT 权限，则该列的数据将以默认值填充。这一点和很多其他的数据库都有一些区别，是 MySQL 自己在 SQL 上面所做的扩展。

5、Routine Level

Routine Level 的权限主要只有 EXECUTE 和 ALTER ROUTINE 两种，主要针对的对象是 procedure 和 function 这两种对象，在授予 Routine Level 权限的时候，需要指定数据库和相关对象，如：

```
root@localhost : test 04:03:26> GRANT EXECUTE ON test.p1 to 'abc'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

除了上面几类权限之外，还有一个非常特殊的权限 GRANT，拥有 GRANT 权限的用户可以

将自身所拥有的任何权限全部授予其他任何用户，所以 GRANT 权限是一个非常特殊也非常重要的权限。GRANT 权限的授予方式也和其他任何权限都不太一样，通常都是通过在执行 GRANT 授权语句的时候在最后添加 WITH GRANT OPTION 子句达到授予 GRANT 权限的目的。

此外，我们还可以通过 GRANT ALL 语句授予某个 Level 的所有可用权限给某个用户，如：

```
root@localhost : test 04:15:48> grant all on test.t5 to 'abc';
Query OK, 0 rows affected (0.00 sec)

root@localhost : test 04:27:39> grant all on perf.* to 'abc';
Query OK, 0 rows affected (0.00 sec)

root@localhost : test 04:27:52> show grants for 'abc';
+------------------------------------------------+
| Grants for abc@%                               |
+------------------------------------------------+
| GRANT USAGE ON *.* TO 'abc'@'%'                |
| GRANT ALL PRIVILEGES ON `perf`.* TO 'abc'@'%'  |
| GRANT ALL PRIVILEGES ON `test`.`t5` TO 'abc'@'%' |
+------------------------------------------------+
```

在以上五个 Level 的权限中，Table、Column 和 Routine 三者在授权中所依赖（或者引用）的对象必须是已经存在的，而不像 Database Level 的权限授予，可以在当前不存在该数据库的时候就完成授权。

4.2.4 MySQL 访问控制实现原理

MySQL 访问控制实际上由两个功能模块共同组成，从第一篇的第二章架构组成中可以看到，一个是负责“看守 MySQL 大门”的用户管理模块，另一个就是负责监控来访者每一个动作的访问控制模块。用户管理模块决定造访客人能否进门，而访问控制模块则决定每个客人进门能拿什么不能拿什么。下面是一张 MySQL 中实现访问控制的简单流程图(见图 4-2)：

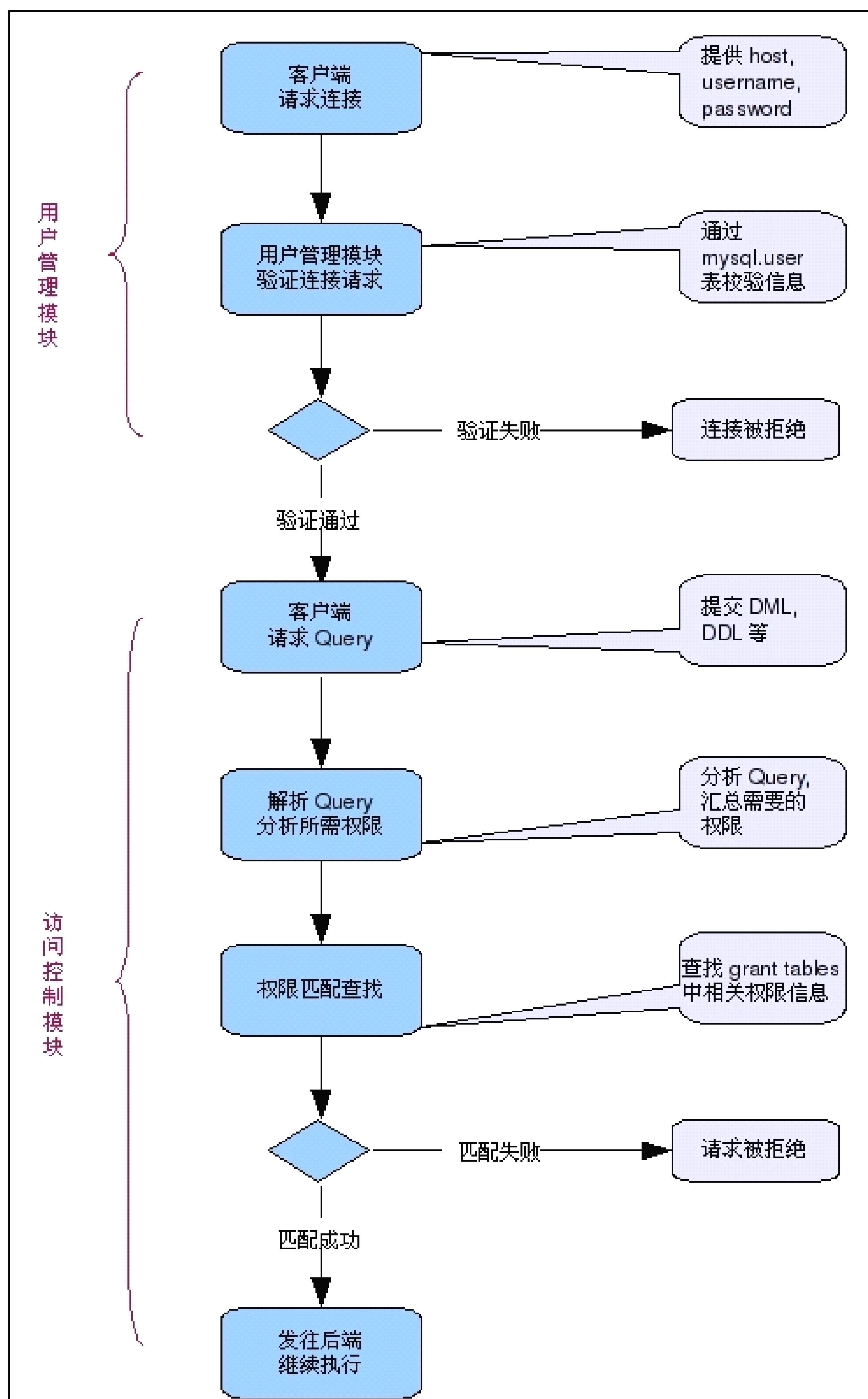

用户管理模块
客户端
请求连接
提供 host,
username,
password
用户管理模块
验证连接请求
通过
mysql.user
表校验信息
验证失败
连接被拒绝
验证通过
访问控制模块
客户端
请求 Query
提交 DML,
DDL 等
解析 Query
分析所需权限
分析 Query,
汇总需要的
权限
权限匹配查找
查找 grant tables
中相关权限信息
匹配失败
请求被拒绝
匹配成功
发往后端
继续执行

图 4-2

1、用户管理

我们先看看用户管理模块是如何工作的。在 MySQL 中，用户访问控制部分的实现比较简单，所有授权用户都存放在一个系统表中：mysql.user，当然这个表不仅仅存放了授权用户的基本信息，还存放有部分细化的权限信息。用户管理模块需要使用的信息很少，主要就是 Host，User，Password 这三项，都在 mysql.user 表中，如下：

```
sky@localhost : (none) 12:35:04> USE mysql;
Database changed
sky@localhost : mysql 12:35:08> DESC user;
+---------------+---------------------+------+-----+---------+-------+
| Field         | Type                | Null | Key | Default | Extra |
+---------------+---------------------+------+-----+---------+-------+
| Host          | char(60)            | NO   | PRI |         |       |
| User          | char(16)            | NO   | PRI |         |       |
| Password      | char(41)            | NO   |     |         |       |
    ... ...
+---------------+---------------------+------+-----+---------+-------+
```

一个用户要想访问 MySQL，至少需要提供上面列出的这三项数据，MySQL 才能判断是否该让他"进门"。这三项实际上由量部分组成：访问者来源的主机名（或者主机 IP 地址信息）和访问者的来访"暗号"（登录用户名和登录密码），这两部分中的任何一个没有能够匹配上都无法让看守大门的用户管理模块乖乖开门。其中 Host 信息存放的是 MySQL 允许所对应的 User 的信任主机，可以是某个具体的主机名（如：mytest）或域名（如：www.domain.com)，也可以是以"%"来充当通配符的某个域名集合（如：%.domain.com)；也可以是一个具体的 IP 地址(如：1.2.3.4)，同样也可以是存在通配符的域名集合(如：1.2.3.%)；还可以用"%"来代表任何主机，就是不对访问者的主机做任何限制。如以下设置：

```
root@localhost : mysql 01:18:12> SELECT host,user,password FROM user ORDER BY user;
+--------------------+------+-------------------------------------------+
| host               | user | password                                  |
+--------------------+------+-------------------------------------------+
| %                  | abc  |                                           |
| *.jianzhaoyang.com | abc  |                                           |
| localhost          | abc  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
| 1.2.3.4            | abc  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
| 1.2.3.*            | def  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
| %                  | def  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
| localhost          | def  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
     ... ...
+--------------------+------+-------------------------------------------+
```

但是这里有一个比较特殊的访问限制，如果要通过 localhost 访问的话，必须要有一条专门针对 localhost 的授权信息，即使不对任何主机做限制也不行。如下例所示，存在 def@% 的用户设置，但是如果不使用-h 参数来访问，则登录会被拒绝，因为 mysql 在默认情况下

会连接 localhost：

```
sky@sky:~$ mysql -u def -p
Enter password:
ERROR 1045 (28000): Access denied for user 'def'@'localhost' (using password: YES)
```

但是当通过-h 参数，明确指定了访问的主机地址之后就没问题了，如下：

```
sky@sky:~$ mysql -u def -p -h 127.0.0.1
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.0.51a-log Source distribution
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
def@127.0.0.1 : (none) 01:26:04>
```

如果我们有一条 localhost 的访问授权则可以不使用-h 参数来指定登录 host 而连接默认的 localhost：

```
sky@sky:~$ mysql -u abc -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18
Server version: 5.0.51a-log Source distribution
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
abc@localhost : (none) 01:27:19> exit
Bye
```

如果 MySQL 正在运行之中的时候，我们对系统做了权限调整，那调整之后的权限什么时候会生效呢？

我们先了解何时 MySQL 存放于内存结构中的权限信息被更新：FLUSH PRIVILEGES 会强行让 MySQL 更新 Load 到内存中的权限信息；GRANT、REVOKE 或者 CREATE USER 和 DROP USER 操作会直接更新内存中俄权限信息；重启 MySQL 会让 MySQL 完全从 grant tables 中读取权限信息。

那内存结构中的权限信息更新之后对已经连接上的用户何时生效呢？

对于 Global Level 的权限信息的修改，仅仅只有更改之后新建连接才会用到，对于已经连接上的 session 并不会受到影响。而对于 Database Level 的权限信息的修改，只有当客户端请求执行了“USE database_name”命令之后，才会在重新校验中使用到新的权限信息。所以有些时候如果在做了比较紧急的 Global 和 Database 这两个 Level 的权限变更之后，可能需要通过“KILL”命令将已经连接在 MySQL 中的 session 杀掉强迫他们重新连接以使用更新后的权限。对于 Table Level 和 Column Level 的权限，则会在下一次需要使用到该权限的 Query 被请求的时候生效，也就是说，对于应用来讲，这两个 Level 的权限，更新之后立刻就生效了，而不会需要执行“KILL”命令。

2、 访问控制

当客户端连接通过用户管理模块的验证，可连接上 MySQL Server 之后，就会发送各种 Query 和 Command 给 MySQL Server，以实现客户端应用的各种功能。当 MySQL 接收到客户端的请求之后，访问控制模块是需要校验该用户是否满足提交的请求所需要的权限。权限校验过程是从最大范围的权限往最小范围的权限开始依次校验所涉及到的每个对象的每个权限。

在验证所有所需权限的时候，MySQL 首先会查找存储在内存结构中的权限数据，首先查找 Global Level 权限，如果所需权限在 Global Level 都有定义（GRANT 或者 REVOKE)，则完成权限校验（通过或者拒绝），如果没有找到所有权限的定义，则会继续往后查找 Database Level 权限，进行 Global Level 未定义的所需权限的校验，如果仍然没有能够找到所有所需权限的定义，MySQL 会继续往更小范围的权限定义域查找，也就是 Table Level，最后则是 Column Level 或者 Routine Level。

下面我们就以客户端通过 abc@localhost 连接后请求如下 Query 我为例：

SELECT id,name FROM test.t4 where status = 'deleted';

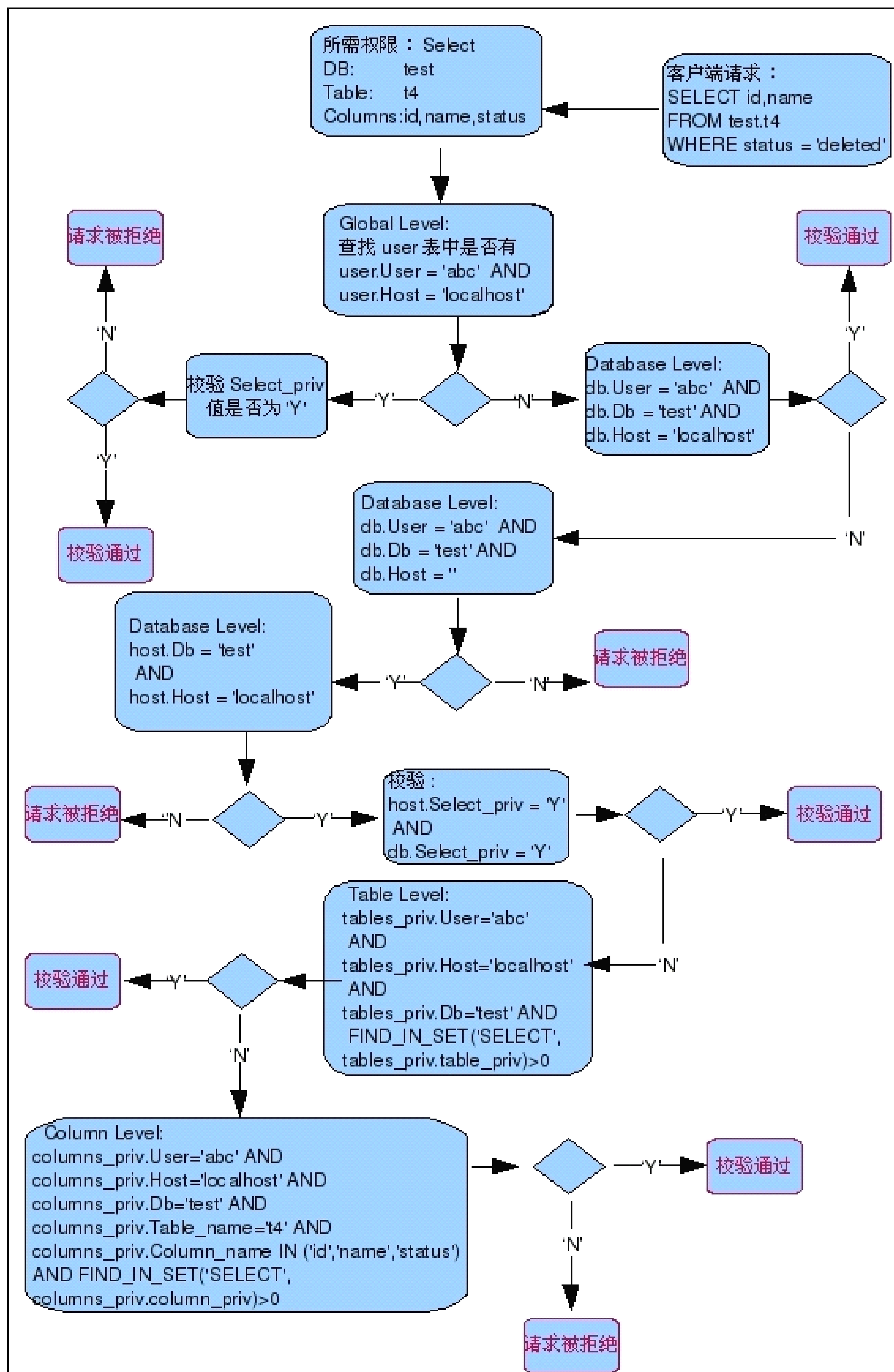


图 4-3

在前面我们了解到 MySQL 的 grant tables 有 mysql.user，mysql.db，mysql.host，mysql.table_priv 和 mysql.column_priv 这五个，我想出了 mysql.host 之外的四个都是非

常容易理解的，每一个表针对MySQL中的一种逻辑对象，存放某一特定Level的权限，唯独mysql.host稍有区别。我们现在就来看看mysql.host权限表到底在MySQL的访问控制中充当了一个什么样的角色呢？

mysql.host在MySQL访问控制模块中所实现的功能比较特殊，和其他几个grant tables不太一样。首先是mysql.host中的权限数据不是（也不能）通过GRANT或者REVOKE来授予或者去除，必须通过手工通过INSERT、UPDATE和DELETE 命令来修改其中的数据。其次是其中的权限数据无法单独生效，必须通过和mysql.db权限表的数据一起才能生效。而且仅当mysql.db中存在不完整（某些场景下的特殊设置）的时候，才会促使访问控制模块再结合mysql.host中查找是否有相应的补充权限数据实现以达到权限校验的目的，就比如上图中所示。在mysql.db中无法找到满足权限校验的所有条件的数据（db.User = 'abc' AND db.host = 'localhost' AND db.Database_name = 'test'），则说明在mysql.db中无法完成权限校验，所以也不会直接就校验db.Select_priv的值是否为'Y'。但是mysql.db中有db.User = 'abc' AND db.Database_name = 'test' AND db.host = '' 这样一条权限信息存在，大家可能注意到了这条权限信息中的db.host中是空值，注意是空值而不是'%'这个通配符哦。当MySQL注意到有这样一条权限信息存在的时候，就该是mysql.host中所存放的权限信息出场的时候了。这时候，MySQL会检测mysql.host中是否存在满足如下条件的权限信息：host.Host = 'localhost' AND host.Db = 'test'。如果存在，则开始进行Select_priv权限的校验。由于权限信息存在于mysql.db和mysql.host两者之中，而且是两者信息合并才能满足要求，所以Select_priv的校验也需要两表都为'Y'才能满足要求，通过校验。

我们已经清楚，MySQL的权限是授予“username@hostname”的，也就是说，至少需要用户名和主机名二者才能确定一个访问者的权限。又由于hostname可以是一个含有通配符的域名，也可以是一个含有通配符的IP地址段。那么如果同一个用户有两条权限信息，一条是针对特定域名的，另外一个是含有通配符的域名，而且前者属于后者包含。这时候MySQL如何来确定权限信息呢？实际上MySQL永远优先考虑更精确范围的权限。在MySQL内部会按照username和hostname作一个排序，对于相同username的权限，其host信息越接近访问者的来源host，则排序位置越靠前，则越早被校验使用到。而且，MySQL在权限校验过程中，只要找到匹配的权限之后，就不会再继续往后查找是否还有匹配的权限信息，而直接完成校验过程。

大家应该也看到了在mysql.user这个权限表中有max_questions，max_updates，max_connections，max_user_connections这四列，前面三列是从MySQL4.0.2版本才开始有的，其功能是对访问用户进行每小时所使用资源的限制，而最后的max_user_connections则是从MySQL5.0.3版本才开始有的，他和max_connections的区别是限制耽搁用户的连接总次数，而不是每小时的连接次数。而要使这四项限制生效，需要在创建用户或者给用户授权的时候加上以下四种子句：

```
max_questions        :    WITH MAX_QUERIES_PER_HOUR n;
max_updates          :    WITH MAX_UPDATES_PER_HOUR n;
max_connections      :    WITH MAX_CONNECTIONS_PER_HOUR n;
max_user_connections:     MAX_USER_CONNECTIONS。
```

四个子句可以同时使用，如：

“ WITH MAX_QUERIES_PER_HOUR 5000 MAX_CONNECTIONS_PER_HOUR 10 MAX_USER_CONNECTIONS 10000”。

4．3 MySQL 访问授权策略

在我们了解了影响数据库系统安全的相关因素以及 MySQL 权限系统的工作原理之后，就需要为我们的系统设计一个安全合理的授权策略。我想，每个人心里都清楚，要想授权最简单最简单方便，维护工作量最少，那自然是将所有权限都授予所有的用户来的最简单方便了。但是，我们大家肯定也都知道，一个用户所用有的权限越大，那么他给我们的系统所带来的潜在威胁也就越大。所以，从安全方面来考虑的话，权限自然是授予的越小越好。一个有足够安全意识的管理员在授权的时候，都会只授予必要的权限，而不会授予任何多余的权限。既然我们这一章是专门讨论安全的，那么我们现在也就从安全的角度来考虑如何设计一个更为安全合理的授权策略。

首先，需要了解来访主机。

由于 MySQL 数据库登录验证用户的时候是出了用户名和密码之外，还要验证来源主机。所以我们还需要了解每个用户可能从哪些主机发起连接。当然，我们也可以通过授权的时候直接通过“%”通配符来给所有主机都有访问的权限，但是这样作就违背了我们安全策略的原则，带来了潜在风险，所以并不可取。尤其是在没有局域网的防火墙保护的情况下，更是不能轻易允许可以从任何主机登录的用户存在。能通过具体主机名或者 IP 地址指定的尽量通过使用具体的主机名和 IP 地址来限定来访主机，不能用具体的主机名或者 IP 地址限定的也需要用尽可能小的通配范围来限定。

其次，了解用户需求。

既然是要做到仅授予必要的权限，那么我们必须了解每个用户所担当的角色，也就是说，我们需要充分了解每个用户需要连接到数据库上完成什么工作。了解该用户是一个只读应用的用户，还是一个读写都有的帐户；是一个备份作业的用户还是一个日常管理的帐户；是只需要访问特定的某个（或者某几个）数据库（Schema)，还是需要访问所有的数据库。只有了解了需要做什么，才能准确的了解需要授予什么样的权限。因为如果权限过低，会造成工作无法正常完成，而权限过高，则存在潜在的安全风险。

再次，要为工作分类。

为了做到各司其职，我们需要将需要做的工作分门别类，不同类别的工作使用不同的用户，做好用户分离。虽然这样可能会带来管理成本方面的部分工作量增加，但是基于安全方面的考虑，这部分管理工作量的增加是非常值得的。而且我们所需要做的用户分离也只是一个适度的分离。比如将执行备份工作、复制工作、常规应用访问、只读应用访问和日常管理工作分别分理出单独的特定帐户来授予各自所需权限。这样，既可以让安全风险尽量降低，也可以让同类同级别的相似权限合并在一起，不互相交织在一起。对于 PROCESS，FILE 和 SUPER 这样的特殊权限，仅仅只有管理类帐号才需要，不应该授予其他非管理帐号。

最后，确保只有绝对必要者拥有 GRANT OPTION 权限。

之前在权限系统介绍的时候我们已经了解到 GRANT OPTION 权限的特殊性，和拥有该权

限之后的潜在风险，所以在这里也就不再累述了。总之，为了安全考虑，拥有 GRANT OPTION 权限的用户越少越好，尽可能只让拥有超级权限的用户才拥有 GRANT OPTION 权限。

4．4 安全设置注意事项

在前面我们了解了影响数据库系统安全的几个因素，也了解了 MySQL 权限系统的相关原理和实现，这一节我们将针对这些因素进行一些基本的安全设置讨论，了解一些必要的注意事项。

首先，自然是最外围第一层防线的网络方面的安全。

我们首先要确定我们所维护的 MySQL 环境是否真的需要提供网络服务？是否可以使我们的 MySQL 仅仅提供本地访问，而禁止网络服务？如果可以，那么我们可以在启动 MySQL 的时候通过使用“--skip-networking”参数选项，让 MySQL 不通过 TCP/IP 监听网络请求，而仅仅通过命名管道或共享内存(在 Windows 中)或 Unix 套接字文件(在 Unix 中)来和客户端连接交互。

当然，在本章最开始的时候，我们就已经讨论过，由于 MySQL 数据库在大部分应用场景中都是在网络环境下，通过网络连接提供服务。所以我们只有少部分应用能通过禁用网络监听来断绝网络访问以保持安全，剩下的大部分还是需要通过其他方案来解决网络方面存在的潜在安全威胁。

使用私有局域网络。我们可以通过使用私有局域网络，通过网络设备，统一私有局域网的出口，并通过网络防火墙设备控制出口的安全。

使用 SSL 加密通道。如果我们的数据对保密要求非常严格，可以启用 MySQL 提供的 SSL 访问接口，将传输数据进行加密。使网络传输的数据即使被截获，也无法轻易使用。

访问授权限定来访主机信息。在之前的权限系统介绍中我们已经了解到 MySQL 的权限信息是针对用户和来访主机二者结合定位的。所以我们可以在授权的时候，通过指定主机的主机名、域名或者 IP 地址信息来限定来访主机的范围。

其次，在第二层防线主机上面也有以下一些需要注意的地方。

OS 安全方面。关闭 MySQL Server 主机上面任何不需要的服务，这不仅能从安全方面减少潜在隐患，还能减轻主机的部分负担，尽可能提高性能。使用网络扫描工具（如 nmap 等）扫描主机端口，检查除了 MySQL 需要监听的端口 3306（或者自定义更改后的某个端口）之外，还有哪些端口是打开正在监听的，并去掉不必要的端口。严格控制 OS 帐号的管理，以防止帐号信息外泄，尤其是 root 和 mysql 帐号。对 root 和 mysql 等对 mysql 的相关文件有特殊操作权限的 OS 帐号登录后做出比较显眼的提示，并在 Terminal 的提示信息中输出当前用户信息，以防止操作的时候经过多次用户切换后出现人为误操作。

用非 root 用户运行 MySQL。这在 MySQL 官方文档中也有非常明显的提示，提醒用户不要使用 root 用户来运行 MySQL。因为如果使用 root 用户运行 MySQL，那么 mysqld 的进程就会拥有 root 用户所拥有的权限，任何具有 FILE 权限的 MySQL 用户就可以在 MySQL 中向系统中的任何位置写入文件。当然，由于 MySQL 不接受操作系统层面的认证，所以任何操作系统

层级的帐号都不能直接登录 MySQL，这一点和 Oracle 的权限认证有些区别，所以在这一方面我们可以减少一些安全方面的顾虑。

文件和进程安全。合理设置文件的权限属性，MySQL 相关的数据和日志文件和所在的文件夹属主和所属组都设置为 mysql，且禁用其他所有用户（除了拥有超级权限的用户，如 root）的读写权限。以防止数据或者日志文件被窃取或破坏。因为如果一个用户对 MySQL 的数据文件有读取权限的话，可以很容易将数据复制。binlog 文件也很容易还原整个数据库。而如果有写权限的话就更糟了，因为有了写权限，数据或者日志文件就有被破坏或者删除的风险存在。保护好 socket 文件的安全，尽量不要使用默认的位置（如/tmp/mysql.sock），以防止被有意或无意的删除。

确保 MySQL Server 所在的主机上所必要运行的其他应用或者服务足够安全，避免因为其他应用或者服务存在安全漏洞而被入侵者攻破防线。

在 OS 层面还有很多关于安全方面的其他设置和需要注意的地方，但考虑到篇幅问题，这里就不做进一步分析了，有兴趣的读者可以参考各种不同 OS 在安全方面的专业书籍。

再次，就是最后第三道防线 MySQL 自身方面的安全设置注意事项。

到了最后这道防线上，我们有更多需要注意的地方。

用户设置。我们必须确保任何可以访问数据库的用户都有一个比较复杂的内容作为密码，而不是非常简单或者比较有规律的字符，以防止被使用字典破解程序攻破。在 MySQL 初始安装完成之后，系统中可能存在一个不需要任何密码的 root 用户，有些版本安装完成之后还会存在一个可以通过 localhost 登录的没有用户名和密码的帐号。这些帐号会给系统带来极大的安全隐患，所以我们必须在正式启用之前尽早删除，或者设置一个比较安全的密码。对于密码数据的存放，也不要存放在简单的文本文件之中，而应该使用专业密码管理软件来管理（如 KeePass）。同时，就像之前在网络安全注意事项部分讲到的那样，尽可能为每一个帐户限定一定范围的可访问主机。尤其是拥有超级权限的 MySQL root 帐号，尽量确保只能通过 localhost 访问。

安全参数。在 MySQL 官方参考手册中也有说明，不论是从安全方面考虑还是从性能以及功能稳定性方面考虑，不需要使用的功能模块尽量都不要启用。例如，如果不需要使用用户自定义函数，就不要在启动的时候使用“--allow-suspicious-udfs”参数选项，以防止被别有居心的潜在威胁者利用此功能而对 MySQL 的安全造成威胁；不需要从本地文件中 Load 数据到数据库中，就使用“--local-infile=0”禁用掉可以从客户端机器上 Load 文件到数据库中；使用新的密码规则和校验规则（不要使用“--old-passwords”启动数据库），这项功能是为了兼容旧版本的密码校验方式的，如无额数必要，不要使用该功能，旧版本的密码加密方式要比新的方式在安全方面弱很多。

除了以上这三道防线，我们还应该让连接 MySQL 数据库的应用程序足够安全，以防止入侵者通过应用程序中的漏洞而入侵到应用服务器，最终通过应用程序中的数据库相关关配置而获取数据库的登录口令。

4．5 小结

安全无小事，一旦安全出了问题一切都完了。数据的安全是一个企业安全方面最核心最重要的内容，只有保障的数据的安全，企业才有可能真正“安全”。希望这一章 MySQL 安全方面的内容能够对各位读者在构筑安全的企业级 MySQL 数据库系统中带来一点帮助。

第 5 章 MySQL 备份与恢复

前言

数据库的备份与恢复一直都是 DBA 工作中最为重要的部分之一，也是基本工作之一。任何正式环境的数据库都必须有完整的备份计划和恢复测试，本章内容将主要介绍 MySQL 数据库的备份与恢复相关内容。

5.1 数据库备份使用场景

你真的明白了自己所做的数据库备份是要面对什么样的场景的吗？

我想任何一位维护过数据库的人都知道数据库是需要备份的，也知道备份数据库是数据库维护必不可少的一件事情。那么是否每一个人都知道自己所做的备份到底是为了应对哪些场景的呢？抑或者说我们每个人是否都很清楚的知道，为什么一个数据库需要作备份呢？读到这里，我想很多读者朋友都会嗤之以鼻，“备份的作用不就是为了防止原数据丢失吗，这谁不知道？”。确实，数据库的备份很大程度上的作用，就是当我们的数据库因为某些原因而造成部分或者全部数据丢失后，方便找回丢失的数据。但是，不同类型的数据库备份，所能应付情况是不一样的，而且，数据库的备份同时也还具有其他很多的作用。而且我想，每个人对数据库备份的作用的理解可能都会有部分区别。

下面我就列举一下我个人理解的我们能够需要用到数据库备份的一些比较常见的情况吧。

一、数据丢失应用场景
1、人为操作失误造成某些数据被误操作；
2、软件 BUG 造成数据部分或者全部丢失；
3、硬件故障造成数据库数据部分或全部丢失；
4、安全漏洞被入侵数据被恶意破坏；

二、非数据丢失应用场景

5、特殊应用场景下基于时间点的数据恢复；
6、开发测试环境数据库搭建；
7、相同数据库的新环境搭建；
8、数据库或者数据迁移；

上面所列出的只是一些常见的应用场景而已，除了上面这几种场景外，数据库备份还会有很多其他应用场景，这里就不一一列举了。那么各位读者过曾经或是现在所做的数据库备份到底是为了应对以上哪一种（或者几种）场景？或者说，我们所做的数据库备份能够应对以上哪几种应用场景？不知道这个问题大家是否有考虑过。

我们必须承认，没有哪一种数据库备份能够解决所有以上列举的几种常见应用场景，即使仅仅只是数据丢失的各种场景都无法通过某一种数据库备份完美的解决，当然也就更不用说能够解决所有的备份应用场景了。

比如当我们遇到磁盘故障，丢失了整个数据库的所有数据，并且无法从已经出现故障的硬盘上面恢复出来的时候，我们可能必须通过一个实时或者有短暂时间差的复制备份数据库存在。当然如果没有这样的一个数据库，就必须要有最近时间的整个数据库的物理或者逻辑备份数据，并且有该备份之后的所有物理或者逻辑增量备份，以期望尽可能将数据恢复到出现故障之前最近的时间点。而当我们遇到认为操作失误造成数据被误操作之后，我们需要有一个能恢复到错误操作时间点之前的瞬间的备份存在，当然这个备份可能是整个数据库的备份，也可以仅仅只是被误操作的表的备份。而当我们要做跨平台的数据库迁移的时候，我们所需要的又只能是一个逻辑的数据库备份，因为平台的差异可能使物理备份的文件格式在两个平台上无法兼容。

既然没有哪一种很多中数据库备份能够完美的解决所有的应用场景，而每个数据库环境所需要面对的数据库备份应用场景又可能各不一样，可能只是需要面对很多种场景中的某一种或几种，那么我们就非常有必要指定一个合适的备份方案和备份策略，通过最简单的技术和最低廉的成本，来满足我们的需求。

5.2 逻辑备份与恢复测试

5.2.1 什么样的备份是数据库逻辑备份呢？

大家都知道，数据库在返回数据给我们使用的时候都是按照我们最初所设计期望的具有一定逻辑关联格式的形式一条一条数据来展现的，具有一定的商业逻辑属性，而在物理存储的层面上数据库软件却是按照数据库软件所设计的某种特定格式经过一定的处理后存放。

数据库逻辑备份就是备份软件按照我们最初所设计的逻辑关系，以数据库的逻辑结构对象为单位，将数据库中的数据按照预定义的逻辑关联格式一条一条生成相关的文本文件，以达到备份的目的。

5.2.2 常用的逻辑备份

逻辑备份可以说是最简单，也是目前中小型系统最常使用的备份方式。在 MySQL 中我们常用的逻辑备份主要就是两种，一种是将数据生成可以完全重现当前数据库中数据的 INSERT 语句，另外一种就是将数据通过逻辑备份软件，将我们数据库表数据以特定分隔符进行分隔后记录在文本文件中。

1、生成 INSERT 语句备份

两种逻辑备份各有优劣，所针对的使用场景也会稍有差别，我们先来看一下生成 INSERT 语句的逻辑备份。

在 MySQL 数据库中，我们一般都是通过 MySQL 数据库软件自带工具程序中的 mysqldump 来实现声称 INSERT 语句的逻辑备份文件。其使用方法基本如下：

```
Dumping definition and data mysql database or table
Usage: mysqldump [OPTIONS] database [tables]
OR     mysqldump [OPTIONS] --databases [OPTIONS] DB1 [DB2 DB3...]
OR     mysqldump [OPTIONS] --all-databases [OPTIONS]
```

由于 mysqldump 的使用方法比较简单，大部分需要的信息都可以通过运行“mysqldump --help”而获得。这里我只想结合 MySQL 数据库的一些概念原理和大家探讨一下当我们使用 mysqldump 来做数据库逻辑备份的时候有些什么技巧以及需要注意一些什么内容。

我们都知道，对于大多数使用数据库的软件或者网站来说，都希望自己数据库能够提供尽可能高的可用性，而不是时不时的就需要停机停止提供服务。因为一旦数据库无法提供服务，系统就无法再通过存取数据来提供一些动态功能。所以对于大多数系统来说如果要让每次备份都停机来做可能都是不可接受的，可是 mysqldump 程序的实现原理是通过我们给的参数信息加上数据库中的系统表信息来一个表一个表获取数据然后生成 INSERT 语句再写入备份文件中的。这样就出现了一个问题，在系统正常运行过程中，很可能会不断有数据变更的请求正在执行，这样就可能造成在 mysqldump 备份出来的数据不一致。也就是说备份数据很可能不是同一个时间点的数据，而且甚至可能都没办法满足完整性约束。这样的备份集对于有些系统来说可能并没有太大问题，但是对于有些对数据的一致性和完整性要求比较严格系统来说问题就大了，就是一个完全无效的备份集。

对于如此场景，我们该如何做？我们知道，想数据库中的数据一致，那么只有两种情况下可以做到。

第一、同一时刻取出所有数据；

第二、数据库中的数据处于静止状态。

对于第一种情况，大家肯定会想，这可能吗？不管如何，只要有两个以上的表，就算我们如何写程序，都不可能昨晚完全一致的取数时间点啊。是的，我们确实无法通过常规方法让取数的时间点完全一致，但是大家不要忘记，在同一个事务中，数据库是可以做到所读取的数据是处于同一个时间点的。所以，对于事务支持的存储引擎，如 Innodb 或者 BDB 等，

我们就可以通过控制将整个备份过程控制在同一个事务中，来达到备份数据的一致性和完整性，而且 mysqldump 程序也给我们提供了相关的参数选项来支持该功能，就是通过“--single-transaction”选项，可以不影响数据库的任何正常服务。

对于第二种情况我想大家首先想到的肯定是将需要备份的表锁定，只允许读取而不允许写入。是的，我们确实只能这么做。我们只能通过一个折衷的处理方式，让数据库在备份过程中仅提供数据的查询服务，锁定写入的服务，来使数据暂时处于一个一致的不会被修改的状态，等 mysqldump 完成备份后再取消写入锁定，重新开始提供完整的服务。mysqldump 程序自己也提供了相关选项如“--lock-tables”和“--lock-all-tables″，在执行之前会锁定表，执行结束后自动释放锁定。这里有一点需要注意的就是，“--lock-tables”并不是一次性将需要 dump 的所有表锁定，而是每次仅仅锁定一个数据库的表，如果你需要 dump 的表分别在多个不同的数据库中，一定要使用“--lock-all-tables”才能确保数据的一致完整性。

当通过 mysqldump 生成 INSERT 语句的逻辑备份文件的时候，有一个非常有用的选项可以供我们使用，那就是“--master-data[=value]″。当添加了“--master-data=1”的时候，mysqldump 会将当前 MySQL 使用到 binlog 日志的名称和位置记录到 dump 文件中，并且是被以 CHANGE_MASTER 语句的形式记录，如果仅仅只是使用“--master-data”或者“--master-data＝2″，则 CHANGE_MASTER 语句会以注释的形式存在。这个选项在实施 slave 的在线搭建的时候是非常有用的，即使不是进行在线搭建 slave，也可以在某些情况下做恢复的过程中通过备份的 binlog 做进一步恢复操作。

在某些场景下，我们可能只是为了将某些特殊的数据导出到其他数据库中，而又不希望通过先建临时表的方式来实现，我们还可以在通过 mysqldump 程序的“—where='where-condition'”来实现，但只能在仅 dump 一个表的情况下使用。

其实除了以上一些使用诀窍之外，mysqldump 还提供了其他很多有用的选项供大家在不同的场景下只用，如通过“--no-data”仅仅 dump 数据库结构创建脚本，通过“--no-create-info″去掉 dump 文件中创建表结构的命令等等，感兴趣的读者朋友可以详细阅读 mysqldump 程序的使用介绍再自行测试。

2、生成特定格式的纯文本备份数据文件备份

除了通过生成 INSERT 命令来做逻辑备份之外，我们还可以通过另外一种方式将数据库中的数据以特定分隔字符将数据分隔记录在文本文件中，以达到逻辑备份的效果。这样的备份数据与 INSERT 命令文件相比，所需要使用的存储空间更小，数据格式更加清晰明确，编辑方便。但是缺点是在同一个备份文件中不能存在多个表的备份数据，没有数据库结构的重建命令。对于备份集需要多个文件，对我们产生的影响无非就是文件多了维护和恢复成本增加，但这些基本上都可以通过编写一些简单的脚本来实现

那我们一般可以使用什么方法来生成这样的备份集文件呢，其实 MySQL 也已经给我们实现的相应的功能。

在 MySQL 中一般都使用以下两种方法来获得可以自定义分隔符的纯文本备份文件。

1、通过执行 SELECT ... TO OUTFILE FROM ...命令来实现

在 MySQL 中提供了一种 SELECT 语法，专供用户通过 SQL 语句将某些特定数据以指定格式输出到文本文件中，同时也提供了实用工具和相关的命令可以方便的将导出文件原样再导入到数据库中。正不正是我们做备份所需要的么？

该命令有几个需要注意的参数如下：

实现字符转义功能的“FIELDS ESCAPED BY ['name']” 将 SQL 语句中需要转义的字符进行转义；

可以将字段的内容“包装”起来的“FIELDS [OPTIONALLY] ENCLOSED BY 'name'”，如果不使用“OPTIONALLY”则包括数字类型的所有类型数据都会被“包装”，使用“OPTIONALLY”之后，则数字类型的数据不会被指定字符“包装”。

通过"FIELDS TERMINATED BY"可以设定每两个字段之间的分隔符；

而通过“LINES TERMINATED BY”则会告诉 MySQL 输出文件在每条记录结束的时候需要添加什么字符。

如以下示例：

```
root@localhost : test 10:02:02> SELECT * INTO OUTFILE '/tmp/dump.text'
    -> FIELDS  TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n'
    -> FROM test_outfile limit 100;
Query OK, 100 rows affected (0.00 sec)

root@localhost : test 10:02:11> exit
Bye
root@sky:/tmp# cat dump.text
350021,21,"A","abcd"
350022,22,"B","abcd"
350023,23,"C","abcd"
350024,24,"D","abcd"
350025,25,"A","abcd"
     ... ...
```

2、通过 mysqldump 导出

可能我们都知道 mysqldump 可以将数据库中的数据以 INSERT 语句的形式生成相关备份文件，其实除了生成 INSERT 语句之外，mysqldump 还同样能实现上面“SELECT ... TO OUTFILE FROM ..."所实现的功能，而且同时还会生成一个相关数据库结构对应的创建脚本。

如以下示例：

```
root@sky:~# ls -l /tmp/mysqldump
total 0
root@sky:~# mysqldump -uroot -T/tmp/mysqldump test test_outfile --fields-enclosed-by=\" --fields-terminated-by=,
root@sky:~# ls -l /tmp/mysqldump
total 8
-rw-r--r-- 1 root  root  1346 2008-10-14 22:18 test_outfile.sql
-rw-rw-rw- 1 mysql mysql 2521 2008-10-14 22:18 test_outfile.txt
```

```
root@sky:~# cat /tmp/mysqldump/test_outfile.txt
350021,21,"A","abcd"
350022,22,"B","abcd"
350023,23,"C","abcd"
350024,24,"D","abcd"
350025,25,"A","abcd"
     ... ...
root@sky:~# cat /tmp/mysqldump/test_outfile.sql
-- MySQL dump 10.11
--
-- Host: localhost    Database: test
-- ------------------------------------------------------
-- Server version   5.0.51a-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `test_outfile`
--

DROP TABLE IF EXISTS `test_outfile`;
SET @saved_cs_client     = @@character_set_client;
SET character_set_client = utf8;
CREATE TABLE `test_outfile` (
  `id` int(11) NOT NULL default '0',
  `t_id` int(11) default NULL,
  `a` char(1) default NULL,
  `mid` varchar(32) default NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
SET character_set_client = @saved_cs_client;

/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

```
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2008-10-14 14:18:23
```

这样的输出结构对我们做为备份来使用是非常合适的，当然如果一次有多个表需要被dump，就会针对每个表都会生成两个相对应的文件。

5.2.3 逻辑备份恢复方法

仅仅有了备份还是不够啊，我们得知道如何去使用这些备份，现在我们就看看上面所做的逻辑备份的恢复方法：

由于所有的备份数据都是以我们最初数据库结构的设计相关的形式所存储，所以逻辑备份的恢复也相对比较简单。当然，针对两种不同的逻辑备份形式，恢复方法也稍有区别。下面我们就分别针对这两种逻辑备份文件的恢复方法做一个简单的介绍。

1、INSERT 语句文件的恢复：

对于 INSERT 语句形式的备份文件的恢复是最简单的，我们仅仅只需要运行该备份文件中的所有（或者部分）SQL 命令即可。首先，如果需要做完全恢复，那么我们可以通过使用“mysql < backup.sql”直接调用备份文件执行其中的所有命令，将数据完全恢复到备份时候的状态。如果已经使用 mysql 连接上了 MySQL，那么也可以通过在 mysql 中执行“source /path/backup.sql”或者“\. /path/backup.sql”来进行恢复。

2、纯数据文本备份的恢复：

如果是上面第二中形式的逻辑备份，恢复起来会稍微麻烦一点，需要一个表一个表通过相关命令来进行恢复，当然如果通过脚本来实现自动多表恢复也是比较方便的。恢复方法也有两个，一是通过 MySQL 的“LOAD DATA INFILE”命令来实现，另一种方法就是通过 MySQL 提供的使用工具 mysqlimport 来进行恢复。

逻辑备份能做什么？不能做什么？

在清楚了如何使用逻辑备份进行相应的恢复之后，我们需要知道我们可以利用这些逻辑备份做些什么。

1、通过逻辑备份，我们可以通过执行相关 SQL 或者命令将数据库中的相关数据完全恢复到备份时候所处的状态，而不影响不相关的数据；

2、通过全库的逻辑备份，我们可以在新的 MySQL 环境下完全重建出一个于备份时候完全一样的数据库，并且不受 MySQL 所处的平台类型限制；

3、通过特定条件的逻辑备份，我们可以将某些特定数据轻松迁移（或者同步）到其他的 MySQL 或者另外的数据库环境；

4、通过逻辑备份，我们可以仅仅恢复备份集中的部分数据而不需要全部恢复。

在知道了逻辑备份能做什么之后，我们必须还要清楚他不能做什么，这样我们自己才能清楚的知道这样的一个备份能否满足自己的预期，是否确实是自己想要的。

1、逻辑备份无法让数据恢复到备份时刻以外的任何一个时刻；

2、逻辑备份无法

5.2.4 逻辑备份恢复测试

时有听到某某的数据库出现问题，而当其信心十足的准备拿之前所做好的数据库进行恢复的时候才发现自己的备份集不可用，或者并不能达到自己做备份时候所预期的恢复效果。遇到这种情景的时候，恐怕每个人都会郁闷至极的。数据库备份最重要最关键的一个用途就是当我们的数据库出现某些异常状况，需要对数据进行恢复的时候使用的。作为一个维护人员，我们是绝对不应该出现此类低级错误的。那我们到底该如何避免此类问题呢？只有一个办法，那就是周期性的进行模拟恢复测试，校验我们的备份集是否真的有效，是否确实能够按照我们的备份预期进行相应的恢复。

到这里可能有人会问，恢复测试又该如何做呢，我们总不能真的将线上环境的数据进行恢复啊？是的，线上环境的数据确实不能被恢复，但是我们为什么不能在测试环境或者其他的地方做呢？做恢复测试只是为了验证我们的备份是否有效，是否能达到我们的预期。所以在做恢复测试之前我们一定要先清楚的知道我们所做的备份到底是为了应用于什么样的场景的。就比如我们做了一个全库的逻辑备份，目的可能是为了当数据库出现逻辑或者物理异常的时候能够恢复整个数据库的数据到备份时刻，那么我们恶的恢复测试就只需要将整个逻辑备份进行全库恢复，看是否能够成功的重建一个完整的数据库。至于恢复的数据是否和备份时刻一致，就只能依靠我们自己来人工判断比较。此外我们可能还希望当某一个数据库对象，比如某个表出现问题之后能够尽快的恢复该表数据到备份时刻。那么我们就可以针对单个指定表进行抽样恢复测试。

下面我们就假想数据库主机崩溃，硬件损坏，造成数据库数据全部丢失，来做一次全库恢复的测试示例：

当我们的数据库出现硬件故障，数据全部丢失之后，我们必须尽快找到一台新的主机以顶替损坏的主机来恢复相应的服务。在恢复服务之前，我们首先需要重建损坏的数据库。假设我们已经拿到了一台新的主机，MySQL 软件也已经安装就位，相关设置也都已经调整好，就等着恢复数据库了。

我们需要取回离崩溃时间最近的一次全库逻辑备份文件，复制到准备的新主机上，启动已经安装好的 MySQL。

由于我们有两种逻辑备份格式，每种格式的恢复方法并不一样，所以这里将对两种格式的逻辑备份的恢复都进行示例。

1、如果是 INSERT 语句的逻辑备份

a、准备好备份文件，copy 到某特定目录，如“/tmp”下；

b、通过执行如下命令执行备份集中的相关命令：

```
mysql -uusername -p < backup.sql
```

或者先通过 mysql 登录到数据库中，然后再执行如下命令：

```
root@localhost : (none) 09:59:40> source /tmp/backup.sql
```

c、再到数据库中检查相应的数据库对象，看是否已经齐全；

d、抽查几个表中的数据进行人工校验，并通知开启应用内部测试校验，当所有校验都通过之后，即可对外提供服务了。

当然上面所说的步骤都是在默认每一步都正常的前提下进行的，如果发现某一步有问题。假若在 b 步骤出现异常，无法继续进行下去，我们首先需要根据出现的错误来排查是否是我们恢复命令有错？是否我们的环境有问题等？等等。如果我们确认是备份文件的问题，那么说明我们的这个备份是无效的，说明测试失败了。如果我们恢复过程很正常，但是在校验的时候发现缺少数据库对象，或者某些对象中的数据不正确，或者根本没有数据。同样说明我们的备份级无法满足预期，备份失败。当然，如果我们是在实际工作的恢复过程中遇到类似情况的时候，如果还有更早的备份集，我们必须退一步使用更早的备份集做相同的恢复操作。虽然更早的备份集中的数据可能会有些失真，但是至少可以部分恢复，而不至于丢失所有数据。

2、如果我们是备份的以特殊分隔符分隔的纯数据文本文件

a、第一步和 INSERT 备份文件没有区别，就是将最接近崩溃时刻的备份文件准备好；

b、通过特定工具或者命令将数据导入如到数据库中：

由于数据库结构创建脚本和纯文本数据备份文件分开存放，所以我们首先需要执行数据库结构创建脚本，然后再导入数据。结构创建脚本的方法和上面第一种备份的恢复测试中的 b 步骤完全一样。

有了数据库结构之后，我们就可以导入备份数据了，如下：

mysqlimport --user=name --password=pwd test --fields-enclosed-by=\″ --fields-terminated-by=, /tmp/test_outfile.txt

或者

LOAD DATA INFILE '/tmp/test_outfile.txt' INTO TABLE test_outfile FIELDS TERMINATED BY '″' ENCLOSED BY ',';

后面的步骤就和备份文件为 INSERT 语句备份的恢复完全一样了，这里就不再累述。

5.3 物理备份与恢复测试

前面一节我们了解了如何使用 MySQL 的逻辑备份，并做了一个简单的逻辑备份恢复示例，在这一节我们再一起了解一些 MySQL 的物理备份。

5.3.1 什么样的备份是数据库物理课备份

在了解 MySQL 的物理备份之前，我们需要先了解一下，什么是数据库物理备份？既然是物理备份，那么肯定是和数据库的物理对象相对应的。就如同逻辑备份根据由我们根据业务逻辑所设计的数据库逻辑对象所做的备份一样，数据库的物理备份就是对数据库的物理对象所做的备份。

数据库的物理对象主要由数据库的物理数据文件、日志文件以及配置文件等组成。在MySQL数据库中，除了MySQL 系统共有的一些日志文件和系统表的数据文件之外，每一种存储引擎自己还会有不太一样的物理对象，在之前第一篇的“MySQL 物理文件组成”中我们已经有了一个基本的介绍，在下面我们将详细列出几种常用的存储引擎各自所对应的物理对象（物理文件），以便在后面大家能够清楚的知道各种存储引擎在做物理备份的时候到底哪些文件是需要备份的哪些又是不需要备份的。

5.3.2 MySQL 物理备份所需文件

MyISAM 存储引擎

MyISAM 存储引擎的所有数据都存放在MySQL配置中所设定的“datadir”目录下。实际上不管我们使用的是 MyISAM 存储引擎还是其他任何存储引擎，每一个数据库都会在“datadir”目录下有一个文件夹（包括系统信息的数据库 mysql 也是一样）。在各个数据库中每一个 MyISAM 存储引擎表都会有三个文件存在，分别为记录表结构元数据的“.frm”文件，存储表数据的“.MYD”文件，以及存储索引数据的“.MYI”文件。由于 MyISAM 属于非事务性存储引擎，所以他没有自己的日志文件。所以 MyISAM 存储引擎的物理备份，除了备份 MySQL 系统的共有物理文件之外，就只需要备份上面的三种文件即可。

Innodb 存储引擎

Innodb 存储引擎属于事务性存储引擎，而且存放数据的位置也可能与 MyISAM 存储引擎有所不同，这主要取决于我们对 Innodb 的“”相关配置所决定。决定 Innodb 存放数据位置的配置为“innodb_data_home_dir”、“innodb_data_file_path”和“innodb_log_group_home_dir”这三个目录位置指定参数，以及另外一个决定 Innodb 的表空间存储方式的参数“innodb_file_per_table″。前面三个参数指定了数据和日志文件的存放位置，最后一个参数决定 Innodb 是以共享表空间存放数据还是以独享表空间方式存储数据。这几个参数的相关使用说明我们已经在第一篇的“MySQL 存储引擎介绍”中做了相应的解释，在 MySQL 的官方手册中也有较为详细的说明，所以这里就不再累述了。

如果我们使用了共享表空间的存储方式，那么 Innodb 需要备份备份“innodb_data_home_dir”和“innodb_data_file_path”参数所设定的所有数据文件，“datadir”中相应数据库目录下的所有 Innodb 存储引擎表的“.frm”文件；

而如果我们使用了独享表空间，那么我们除了备份上面共享表空间方式所需要备份的所有文件之外，我们还需要备份“datadir”中相应数据库目录下的所有“.idb”文件，该文件中存放的才是独享表空间方式下 Innodb 存储引擎表的数据。可能在这里有人文，既然是使用独享表空间，那我们为什么还要备份共享表空间“才使用到”的数据文件呢？其实这是很多人的一个共性误区，以为使用独享表空间的时候 Innodb 的所有信息就都存放在“datadir”所设定数据库目录下的“.ibd”文件中。实际上并不是这样的，“.ibd”文件中所存放的仅仅只是我们的表数据而已，大家都很清楚，Innodb 是事务性存储引擎，他是需要 undo 和 redo 信息的，而不管 Innodb 使用的是共享还是独享表空间的方式来存储数据，与事务相关的 undo 信息以及其他的一些元数据信息，都是存放在“innodb_data_home_dir”

和“innodb_data_file_path”这两个参数所设定的数据文件中的。所以要想Innodb的物理备份有效，“innodb_data_home_dir”和“innodb_data_file_path”参数所设定的数据文件不管在什么情况下我们都必须备份。

此外，除了上面所说的数据文件之外，Innodb还有自己存放redo信息和相关事务信息的日志文件在“innodb_log_group_home_dir”参数所设定的位置。所以要想Innodb物理备份能够有效使用，我们还比需要备份“innodb_log_group_home_dir”参数所设定的位置的所有日志文件。

NDB Cluster存储引擎

NDB Cluster 存储引擎（其实也可以说是 MySQL Cluster）的物理备份需要备份的文件主要有一下三类：

1、 元数据（Metadata）：包含所有的数据库以及表的定义信息；
2、 表数据（Table Records）：保存实际数据的文件；
3、 事务日志数据（Transaction Log）：维持事务一致性和完整性，以及恢复过程中所需要的事务信息。

不论是通过停机冷备份，还是通过 NDB Cluster 自行提供的在线联机备份工具，或者是第三方备份软件来进行备份，都需要备份以上三种物理文件才能构成一个完整有效的备份集。当然，相关的配置文件，尤其是管理节点上面的配置信息，同样也需要备份。

5.3.3 各存储引擎常用物理备份方法

由于不同存储引擎所需要备份的物理对象（文件）并不一样，且每个存储引擎对数据文件的一致性要求也不一样所以各个存储引擎在进行物理备份的时候所使用的备份方法也有区别。当然，如果我们是要做冷备份（停掉数据库之后的备份），我们所需要做的事情都很简单，那就是直接copy所有数据文件和日志文件到备份集需要存放的位置即可，不管是何种存储引擎都可以这样做。由于冷备份方法简单，实现容易，所以这里就不详细说明了。

在我们的实际应用环境中，是很少有能够让我们可以停机做日常备份的情况的，我们只能在数据库提供服务的情况下来完成数据库备份。这也就是我们俗称的热物理备份了。下面我们就针对各个存储引擎单独说明各自最常用的在线（热）物理备份方法。

MyISAM存储引擎

上面我们介绍了MyISAM存储引擎文件的物理文件比较集中，而且不支持事务没有redo和undo日志，对数据一致性的要求也并不是特别的高，所以MyISAM存储引擎表的物理备份也比较简单，只要将MyISAM的物理文件copy出来即可。但是，虽然MyISAM存储引擎没有事务支持，对数据文件的一致性要求没有 Innodb之类的存储引擎那么严格，但是 MyISAM存储引擎的同一个表的数据文件和索引文件之间是有一致性要求的。当MyISAM存储引擎发现某个表的数据文件和索引文件不一致的时候，会标记该表处于不可用状态，并要求你进行修复动作，当然，一般情况下的修复都会比较容易。但是，即使数据库存储引擎本身对数据文件的一致性要求并不是很苛刻，我们的应用也允许数据不一致吗？我想答案肯定是否定

的，所以我们自己必须至少保证数据库在备份时候的数据是处于某一个时间点的，这样就要求我们必须做到在备份 MyISAM 数据库的物理文件的时候让 MyISAM 存储引擎停止写操作，仅仅提供读服务，其根本实质就是给数据库表加锁来阻止写操作。

MySQL 自己提供了一个使用程序 mysqlhotcopy，这个程序就是专门用来备份 MyISAM 存储引擎的。不过如果你有除了 MyISAM 之外的其他非事务性存储引擎，也可以通过合适的参数设置，或者微调该备份脚本，也都能通过 mysqlhotcopy 程序来完成相应的备份任务，基本用法如下：

```
mysqlhotcopy db_name[./table_regex/] [new_db_name | directory]
```

从上面的基本使用方法我们可以看到，mysqlhotcopy 出了可以备份整个数据库，指定的某个表，还可以通过正则表达式来匹配某些表名来针对性的备份某些表。备份结果就是指定数据库的文件夹下包括所有指定的表的相应物理文件。

mysqlhotcopy 是一个用 perl 编写的使用程序，其主要实现原理实际上就是通过先 LOCK 住表，然后执行 FLUSH TABLES 动作，该正常关闭的表正常关闭，将该 fsync 的数据都 fsync，然后通过执行 OS 级别的复制（cp 等）命令，将需要备份的表或者数据库的所有物理文件都复制到指定的备份集位置。

此外，我们也可以通过登录数据库中手工加锁，然后再通过操作系统的命令来复制相关文件执行热物理备份，且在完成文件 copy 之前，不能退出加锁的 session（因为退出会自动解锁），如下：

```
root@localhost : test 08:36:35> FLUSH TABLES WITH READ LOCK;
Query OK, 0 rows affected (0.00 sec)
```

不退出 mysql，在新的终端下做如下备份：

```
mysql@sky:/data/mysql/mydata$ cp -R test /tmp/backup/test
mysql@sky:/data/mysql/mydata$ ls -l /tmp/backup/
total 4
drwxr-xr-x 2 mysql mysql 4096 2008-10-19 21:57 test
mysql@sky:/data/mysql/mydata$ ls -l /tmp/backup/test
total 39268
-rw-r----- 1 mysql mysql     8658 2008-10-19 21:57 hotcopy_his.frm
-rw-r----- 1 mysql mysql       36 2008-10-19 21:57 hotcopy_his.MYD
-rw-r----- 1 mysql mysql     1024 2008-10-19 21:57 hotcopy_his.MYI
-rw-r----- 1 mysql mysql     8586 2008-10-19 21:57 memo_test.frm
    ... ...
-rw-rw---- 1 mysql mysql     8554 2008-10-19 22:01 test_csv.frm
-rw-rw---- 1 mysql mysql        0 2008-10-19 22:01 test_csv.MYD
-rw-rw---- 1 mysql mysql     1024 2008-10-19 22:01 test_csv.MYI
-rw-r----- 1 mysql mysql     8638 2008-10-19 21:57 test_myisam.frm
```

```
-rw-r----- 1 mysql mysql 20999600 2008-10-19 21:57 test_myisam.MYD
-rw-r----- 1 mysql mysql 10792960 2008-10-19 21:57 test_myisam.MYI
-rw-r----- 1 mysql mysql     8638 2008-10-19 21:57 test_outfile.frm
-rw-r----- 1 mysql mysql     2400 2008-10-19 21:57 test_outfile.MYD
-rw-r----- 1 mysql mysql     1024 2008-10-19 21:57 test_outfile.MYI
    ... ...
```

然后再在之前的执行锁定命令的 session 中解锁

```
root@localhost : test 10:00:57> unlock tables;
Query OK, 0 rows affected (0.00 sec)
```

这样就完成了一次物理备份，而且大家也从文件列表中看到了，备份中还有 CSV 存储引擎的表。

Innodb 存储引擎

Innodb 存储引擎由于是事务性存储引擎，有 redo 日志和相关的 undo 信息，而且对数据的一致性和完整性的要求也比 MyISAM 要严格很多，所以 Innodb 的在线（热）物理备份要比 MyISAM 复杂很多，一般很难简单的通过几个手工命令来完成，大都是通过专门的 Innodb 在线物理备份软件来完成。

Innodb 存储引擎的开发者(Innobase 公司)开发了一款名为 ibbackup 的商业备份软件，专门实现 Innodb 存储引擎数据的在线物理备份功能。该软件可以在 MySQL 在线运行的状态下，对数据库中使用 Innodb 存储引擎的表进行备份，不过仅限于使用 Innodb 存储引擎的表。

由于这款软件并不是开源免费的产品，我个人也很少使用，主要也是下载的试用版试用而已，所以这里就不详细介绍了，各位读者朋友可以通过 Innobase 公司官方网站获取详细的使用手册进行试用

NDB Cluster 存储引擎

NDB Cluster 存储引擎也是一款事务性存储引擎，和 Innodb 一样也有 redo 日志。NDB Cluter 存储引擎自己提供了备份功能，可以通过相关的命令实现。当然，停机冷备的方法也是有效的。

在线联机备份步骤如下：

1、 连接上管理服务器；
2、 在管理节点上面执行 “START BACKUP” 命令；
3、 在管理节点上发出备份指令之后，管理节点会通知所有数据节点开始进行备份，并反馈通知结果。
4、 管理节点在通知发出备份指令之前会生成一个备份号来唯一定位这次备份所产生的备份集。当各数据节点收到备份指令之后，就会开始进行备份操作。
5、 当所有数据节点都完成备份之后，管理节点才会反馈“备份完成”的信息给客户端。

由于 NDB Cluster 的备份，备份指令是从管理节点发起，且并不会等待备份完成就会返回，所以也没办法直接通过 “Ctrl + c” 或者其他方式来中断备份进程，所以 NDB Cluster 提供了相应的命令来中断当前正在进行的备份操作，如下：

1、 登录管理节点
2、 执行 “ABORT BACKUP backup_id"，命令中的 backup_id 即之前发起备份命令的时候所产生的备份号。
3、 管理结带你上会用消息“放弃指示的备份 backup_id”确认放弃请求，注意，则时候其实并没有收到数据节点对请求的实际回应。
4、 然后管理节点才会将中断备份的指令发送到所有数据节点上面，然后当各个数据节点都中断备份并删除了当前产生的备份文件之后，才会返回“备份 backup_id 因＊＊＊而放弃"。至此，中断备份操作完成。

通过 NDB Cluster 存储引擎自己的备份命令来进行备份之后，会将前面所提到的三种文件存放在参与备份的节点上面，且被存放在三个不同的文件中，类似如下：

BACKUP-backup_id.node_id.ctl，内容包含相关的控制信息和元数据的控制文件。每个节点均会将相同的表定义（对于 Cluster 中的所有表）保存在自己的该文件中。

BACKUP-backup_id-n.node_id.data，数据备份文件，被分成多个不同的片段来保存，在备份过程中，不同的节点将保存不同的备份数据所产生的片段，每个节点保存的文件都会有信息指明数据所属表的部分，且在备份片段文件最后还包含了最后的校验信息，以确保备份能够正确恢复。

BACKUP-backup_id.node_id.log，事务日志备份文件中仅包含已提交事务的相关信息，且仅保存已在备份中保存的表上的事务，各个阶段所保存的日志信息也不一样，因为仅仅针对各节点所包含的数据记录相关的日志信息。

上面的备份文件命名规则中，backup_id 是指备份号，不同的备份集会针对有一个不同的备份号，node_id 则是指明该备份文件属于哪个数据节点，而在数据文件的备份文件中的 n 则是指明片段号。

5.3.4 各存储引擎常用物理备份恢复方法

和之前逻辑备份一样，光有备份是没有意义的，还需要能够将备份有效的恢复才行。物理备份和逻辑备份相比最大的优势就是恢复速度快，因为主要是物理文件的拷贝，将备份文件拷贝到需要恢复的位置，然后进行简单的才做即可。

MyISAM 存储引擎

MyISAM 存储引擎由于其特性，物理备份的恢复也比较简单。

如果是通过停机冷备份或者是在运行状态通过锁定写入操作后的备份集来恢复，仅仅只需要将该备份集直接通过操作系统的拷贝命令将相应的数据文件复制到对应位置来覆盖现有文件即可。

如果是通过 mysqlhotcopy 软件来进行的在线热备份，而且相关的备份信息也记录进入了数据库中相应的表，其恢复操作可能会需要结合备份表信息来进行恢复。

Innodb 存储引擎

对于冷备份，Innodb 存储引擎进行恢复所需要的操作和其他存储引擎没有什么差别，同样是备份集文件（包括数据文件和日志文件）复制到相应的目录即可。但是对于通过其他备份软件所进行的备份，就需要根据备份软件本身的要求来进行了。比如通过 ibbackup 来进行的备份，同样也需要通过他来进行恢复才可以，具体的恢复方法请通过该软件的使用手册来进行，这里就不详细介绍了。

NDB Cluster 存储引擎

对于停机冷备，恢复方法和其他存储引擎也没有太多区别，只不过有一点需要特别注意的就是恢复的时候必须要将备份集中文件恢复到对应的数据节点之少，否则无法正确完成恢复过程。

而通过 NDB Cluster 所提供的备份命令来生成的备份集，需要使用专用的备份恢复软件 ndb_restore 来进行。ndb_restore 软件将从备份集中读取出备份相关的控制信息，而且 ndb_restore 软件必须在单独的数据节点上面分别进行。所以当初备份进行过程中有多少数据节点，现在就需要运行多少次 ndb_restore。而且，首次通过 ndb_restore 来进行恢复的话，还必须恢复元数据，也就是会重建所有的数据库和表。

5.5 备份策略的设计思路

备份是否完整，能否满足要求，关键还是需要看所设计的备份策略是否合理，以及备份操作是否确实按照所设计的备份策略进行了。

针对于不同的用途，所需要的备份类型是不一样的，所以需要的备份策略有各有不同。如为了应对本章最开始所描述的在线应用的数据丢失的问题，我们的备份就需要快速恢复，而且最好是仅仅需要增量恢复就能找回所需数据。对于这类需求，最好是有在线的，且部分延迟恢复的备用数据库。因为这样可以在最短时间内找回所需要的数据。甚至在某些硬件设备出现故障的时候，将备用库直接开发对外提供服务都可以。当然，在资源缺乏的情况下，可能难以找到足够的备用硬件设备来承担这个备份责任的时候，我们也可以通过物理备份来解决，毕竟物理备份的恢复速度要比逻辑备份的快很多。

而对于那些非数据丢失的应用场景，大多数时候恢复时间的要求并不是太高，只要可以恢复出一个完整可用的数据库就可以了。所以不论是物理备份还是逻辑备份，影响都不大。

从我个人经验来看，可以根据不同的需求不同的级别通过如下的几个思路来设计出合理的备份策略：

1、 对于较为核心的在线应用系统，比需要有在线备用主机通过 MySQL 的复制进行相

应的备份，复制线程可以一直开启，恢复线程可以每天恢复一次，尽量让备机的数据延后主机在一定的时间段之内。这个延后的时间多长合适主要是根据实际需求决定，一般来说延后一天是一个比较常规的做法。

2、 对于重要级别稍微低一些的应用，恢复时间要求不是太高的话，为了节约硬件成本，不必要使用在线的备份主机来单独运行备用 MySQL，而是通过每一定的时间周期内进行一次物理全备份，同时每小时（或者其他合适的时间段）内将产生的二进制日志进行备份。这样虽然没有第一种备份方法恢复快，但是数据的丢失会比较少。恢复所需要的时间由全备周期长短所决定。
3、 而对于恢复基本没有太多时间要求，但是不希望太多数据丢失的应用场景，则可以通过每一定时间周期内进行一次逻辑全备份，同时也备份相应的二进制日志。使用逻辑备份而不使用物理备份的原因是因为逻辑备份实现简单，可以完全在线联机完成，备份过程不会影响应用提供服务。
4、 对于一些搭建临时数据库的备份应用场景，则仅仅只需要通过一个逻辑全备份即可满足需求，都不需要用二进制日志来进行恢复，因为这样的需求对数据并没有太苛刻的要求。

上面的四种备份策略都还比较较粗糙，甚至不能算是一个备份策略。目的只是希望能给大家一个指定备份策略的思路。各位读者朋友可以根据这个思路根据实际的应用场景，指定出各种不同的备份策略。

5.6 小结

总的来说，MySQL 的备份与恢复都不是太复杂，方法也比较单一。姑且不说逻辑备份，对于物理备份来说，确实是还不够完善。缺少一个开源的比较好的在线热物理备份软件，一直是 MySQL 一个比较大的遗憾，也是所有 MySQL 使用者比较郁闷的事情。

当然，没有开源的备份软件使用，非开源的商业软件也还是有的，如比较著名的 Zmanda 备份恢复软件，功能就比较全面，使用也不太复杂，在商业的 MySQL 备份恢复软件市场上有较高的占有率。而且，Zmanda 同时还提供社区版本的免费下载使用。

不过，稍微让人有所安慰的是 MySQL 在实际应用场景中大多是有一台或者多台 Slave 机器来作为热备的。在需要进行备份的时候通过 Slave 来进行备份也不是太难，而且通过暂时停止 Slave 上面的 SQL 线程，即可让 Slave 机器停止所有数据写入操作，然后就可以进行在线进行备份操作了。所以即使买不起商用软件或者不太想买关系也不是太大。

第 6 章 影响 MySQL Server 性能的相关因素

前言：

大部分人都一致认为一个数据库应用系统（这里的数据库应用系统概指所有使用数据库的系统）的性能瓶颈最容易出现在数据的操作方面，而数据库应用系统的大部分数据操作都是通过数据库管理软件所提供的相关接口来完成的。所以数据库管理软件也就很自然的成为了数据库应用系统的性能瓶颈所在，这是当前业界比较普遍的一个看法。但我们的应用系统的性能瓶颈真的完全是因为数据库管理软件和数据库主机自身造成的吗？我们将通过本章的内容来进行一个较为深入的分析，让大家了解到一个数据库应用系统的性能到底与哪些地方有关，让大家寻找出各自应用系统的出现性能问题的根本原因，而尽可能清楚的知道该如何去优化自己的应用系统。

考虑到本书的数据库对象是 MySQL，而 MySQL 最多的使用场景是 WEB 应用，那么我们就以一个 WEB 应用系统为例，逐个分析其系统构成，结合笔者在大型互联网公司从事 DBA 工作多年的经验总结，分析出数据库应用系统中各个环境对性能的影响。

6.1 商业需求对性能的影响

应用系统中的每一个功能在设计初衷肯定都是出于为用户提供某种服务，或者满足用户的某种需求，但是，并不是每一个功能在最后都能很成功，甚至有些功能的推出可能在整个系统中是画蛇添足。不仅没有为用户提高任何体验度，也没有为用户改进多少功能易用性，反而在整个系统中成为一个累赘，带来资源的浪费。

不合理需求造成资源投入产出比过低

需求是否合理很多时候可能并不是很容易界定，尤其是作为技术人员来说，可能更难以确定一个需求的合理性。即使指出，也不一定会被产品经历们认可。那作为技术人员的我们怎么来证明一个需求是否合理呢？

第一、每次产品经理们提出新的项目（或者功能需求）的时候，应该要求他们同时给出该项目的预期收益的量化指标，以备项目上先后统计评估投入产出比率；

第二、在每次项目进行过程中，应该详细记录所有的资源投入，包括人力投入，硬件设施的投入，以及其他任何项目相关的资源投入；

第三、项目（或者功能需求）上线之后应该及时通过手机相关数据统计出项目的实际收益值，以便计算投入产出比率的时候使用；

第四、技术部门应该尽可能推动设计出一个项目（或者功能需求）的投入产出比率的计算规则。在项目上线一段时间之后，通过项目实际收益的统计数据和项目的投入资源量，计算出整个项目的实际投入产出值，并公布给所有参与项目的部门知晓，同时存放以备后查。

有了实际的投入产出比率，我们就可以和项目立项之初产品经理们的预期投入产出比率做出比较，判定出这个项目做的是否值得。而且当积累了较多的项目投入产出比率之后，我们可以根据历史数据分

析出一个项目合理的投入产出比率应该是多少。这样，在项目立项之初，我们就可以判定出产品经理们的预期投入产出比率是否合理，项目是否真的有进行的必要。

有了实际的投入产出比率之后，我们还可以拿出数据给老板们看，让他知道功能并不是越多越好，让他知道有些功能是应该撤下来的，即使撤下该功能可能需要投入不少资源。

实际上，一般来说，在产品开发及运营部门内部都会做上面所说的这些事情的。但很多时候可能更多只是一种形式化的过程。在有些比较规范的公司可能也完成了上面的大部分流程，但是要么数据不公开，要么公开给其他部门的数据存在一定的偏差，不具备真实性。

为什么会这样？其实就一个原因，就是部门之间的利益冲突及业绩冲突问题。产品经理们总是希望尽可能的让用户觉得自己设计的产品功能齐全，让老板觉得自己做了很多事情。但是从来都不会去关心因为做一个功能所带来的成本投入，或者说是不会特别的关心这一点。而且很多时候他们也并不能太理解技术方面带来的复杂度给产品本身带来的负面影响。

这里我们就拿一个看上去很简单的功能来分析一下。
需求：一个论坛帖子总量的统计
附加要求：实时更新

在很多人看来，这个功能非常容易实现，不就是执行一条 SELECT COUNT(*)的 Query 就可以得到结果了么？是的，确实只需要如此简单的一个 Query 就可以得到结果。但是，如果我们采用不是 MyISAM 存储引擎，而是使用的 Innodb 的存储引擎，那么大家可以试想一下，如果存放帖子的表中已经有上千万的帖子的时候，执行这条 Query 语句需要多少成本？恐怕再好的硬件设备，恐怕都不可能在 10 秒之内完成一次查询吧。如果我们的访问量再大一点，还有人觉得这是一件简单的事情么？

既然这样查询不行，那我们是不是该专门为这个功能建一个表，就只有一个字段，一条记录，就存放这个统计量，每次有新的帖子产生的时候，都将这个值增加 1，这样我们每次都只需要查询这个表就可以得到结果了，这个效率肯定能够满足要求了。确实，查询效率肯定能够满足要求，可是如果我们的系统帖子产生很快，在高峰时期可能每秒就有几十甚至上百个帖子新增操作的时候，恐怕这个统计表又要成为大家的噩梦了。要么因为并发的问题造成统计结果的不准确，要么因为锁资源争用严重造成整体性能的大幅度下降。

其实这里问题的焦点不应该是实现这个功能的技术细节，而是在于这个功能的附加要求“实时更新”上面。当一个论坛的帖子数量很大了之后，到底有多少人会关注这个统计数据是否是实时变化的？有多少人在乎这个数据在短时间内的不精确性？我想恐怕不会有人会傻傻的盯着这个统计数字并追究当自己发了一个帖子然后回头刷新页面发现这个统计数字没有加 1 吧？即使明明白白的告诉用户这个统计数据是每过多长时间段更新一次，那有怎样？难道会有很多用户就此很不爽么？

只要去掉了这个“实时更新”的附加条件，我们就可以非常容易的实现这个功能了。就像之前所提到的那样，通过创建一个统计表，然后通过一个定时任务每隔一定时间段去更新一次里面的统计值，这样既可以解决统计值查询的效率问题，又可以保证不影响新发贴的效率，一举两得。

实际上，在我们应用的系统中还有很多很多类似的功能点可以优化。如某些场合的列表页面参与列表的数据量达到一个数量级之后，完全可以不用准确的显示这个列表总共有多少条信息，总共分了多少

页，而只需要一个大概的估计值或者一个时间段之前的统计值。这样就省略了我们的分页程序需要在分以前实时 COUNT 出满足条件的记录数。

其实，在很多应用系统中，实时和准实时，精确与基本准确，在很多地方所带来的性能消耗可能是几个性能的差别。在系统性能优化中，应该尽量分析出那些可以不实时和不完全精确的地方，作出一些相应的调整，可能会给大家带来意想不到的巨大性能提升。

无用功能堆积使系统过度复杂影响整体性能

很多时候，为系统增加某个功能可能并不需要花费太多的成本，而要想将一个已经运行了一段时间的功能从原有系统中撤下来却是非常困难的。

首先，对于开发部门，可能要重新整理很多的代码，找出可能存在与增加该功能所编写的代码有交集的其他功能点，删除没有关联的代码，修改有关联的代码；
其次，对于测试部门，由于功能的变动，必须要回归测试所有相关的功能点是否正常。可能由于界定困难，不得不将回归范围扩展到很大，测试工作量也很大。
最后，所有与撤除下线某个功能相关的工作参与者来说，又无法带来任何实质性的收益，而恰恰相反是，带来的只可能是风险。

由于上面的这几个因素，可能很少有公司能够有很完善的项目（或者功能）下线机制，也很少有公司能做到及时将系统中某些不合适的功能下线。所以，我们所面对的应用系统可能总是越来越复杂，越来越庞大，短期内的复杂可能并无太大问题，但是随着时间的积累，我们所面对的系统就会变得极其臃肿。不仅维护困难，性能也会越来越差。尤其是有些并不合理的功能，在设计之初或者是刚上线的时候由于数据量较小，带来不了多少性能损耗。可随着时间的推移，数据库中的数据量越来越大，数据检索越来越困难，对真个系统带来的资源消耗也就越来越大。

而且，由于系统复杂度的不断增加，给后续其他功能的开发带来实现的复杂度，可能很多本来很简单的功能，因为系统的复杂而不得不增加很多的逻辑判断，造成系统应用程序的计算量不断增加，本身性能就会受到影响。而如果这些逻辑判断还需要与数据库交互通过持久化的数据来完成的话，所带来的性能损失就更大，对整个系统的性能影响也就更大了。

6.2 系统架构及实现对性能的影响

一个 WEB 应用系统，自然离不开 Web 应用程序（Web App）和应用程序服务器（App Server）。App Server 我们能控制的内容不多，大多都是使用已经久经考验的成熟产品，大家能做的也就只是通过一些简单的参数设置调整来进行调优，不做细究。而 Web App 大部分都是各自公司根据业务需求自行开发，可控性要好很多。所以我们从 Web 应用程序着手分析一个应用程序架构的不同设计对整个系统性能的影响将会更合适。

上一节中商业需求告诉了我们一个系统应该有什么不应该有什么，系统架构则则决定了我们系统的构建环境。就像修建一栋房子一样，在清楚了这栋房子的用途之后，会先有建筑设计师来画出一章基本

的造型图，然后还需要结构设计师为我们设计出结构图。系统架构设计的过程就和结构工程好似设计结构图一样，需要为整个系统搭建出一个尽可能最优的框架，让整个系统能够有一个稳定高效的结构体系让我们实现各种商业需求。

谈到应用系统架构的设计，可能有人的心里会开始嘀咕，一个 DBA 有什么资格谈论人家架构师（或者程序员）所设计的架构？其实大家完全没有必要这样去考虑，我们谈论架构只是分析各种情形下的性能消耗区别，仅仅是根据自己的专业特长来针对相应架构给出我们的建议及意见，并不是要批判架构整体的好坏，更不是为了推翻某个架构。而且我们所考虑的架构大多数时候也只是数据层面相关的架构。

我们数据库中存放的数据都是适合在数据库中存放的吗？

对于有些开发人员来说，数据库就是一个操作最方便的万能存储中心，希望什么数据都存放在数据库中，不论是需要持久化的数据，还是临时存放的过程数据，不论是普通的纯文本格式的字符数据，还是多媒体的二进制数据，都喜欢全部塞如数据库中。因为对于应用服务器来说，数据库很多时候都是一个集中式的存储环境，不像应用服务器那样可能有很多台；而且数据库有专门的 DBA 去帮忙维护，而不像应用服务器很多时候还需要开发人员去做一些维护；还有一点很关键的就是数据库的操作非常简单统一，不像文件操作或者其他类型的存储方式那么复杂。

其实我个人认为，现在的很多数据库为我们提供了太多的功能，反而让很多并不是太了解数据库的人错误的使用了数据库的很多并不是太擅长或者对性能影响很大的功能，最后却全部怪罪到数据库身上。

实际上，以下几类数据都是不适合在数据库中存放的：

1. 二进制多媒体数据

将二进制多媒体数据存放在数据库中，一个问题是数据库空间资源耗用非常严重，另一个问题是这些数据的存储很消耗数据库主机的 CPU 资源。这种数据主要包括图片，音频、视频和其他一些相关的二进制文件。这些数据的处理本不是数据的优势，如果我们硬要将他们塞入数据库，肯定会造成数据库的处理资源消耗严重。

2. 流水队列数据

我们都知道，数据库为了保证事务的安全性（支持事务的存储引擎）以及可恢复性，都是需要记录所有变更的日志信息的。而流水队列数据的用途就决定了存放这种数据的表中的数据会不断的被 INSERT，UPDATE 和 DELETE，而每一个操作都会生成与之对应的日志信息。在 MySQL 中，如果是支持事务的存储引擎，这个日志的产生量更是要翻倍。而如果我们通过一些成熟的第三方队列软件来实现这个 Queue 数据的处理功能，性能将会成倍的提升。

3. 超大文本数据

对于 5.0.3 之前的 MySQL 版本，VARCHAR 类型的数据最长只能存放 255 个字节，如果需要存储更长的文本数据到一个字段，我们就必须使用 TEXT 类型（最大可存放 64KB）的字段，甚至是更大的 LONGTEXT 类型（最大 4GB）。而 TEXT 类型数据的处理性能要远比 VARCHAR 类型数据的处理性能低下很多。从 5.0.3 版本开始，VARCHAR 类型的最大长度被调整到 64KB 了，但是当实际数据小于 255 Bytes 的时候，实际存储空间和实际的数据长度一样，可一旦长度超过 255 Bytes 之后，所占用的存储空间就是实际数据长度的两倍。

所以，超大文本数据存放在数据库中不仅会带来性能低下的问题，还会带来空间占用的浪费问题。

是否合理的利用了应用层Cache机制?

对于Web应用，活跃数据的数据量总是不会特别的大，有些活跃数据更是很少变化。对于这类数据，我们是否有必要每次需要的时候都到数据库中去查询呢？如果我们能够将变化相对较少的部分活跃数据通过应用层的Cache机制Cache到内存中，对性能的提升肯定是成数量级的，而且由于是活跃数据，对系统整体的性能影响也会很大。

当然，通过Cache机制成功的案例数不胜数，但是失败的案例也同样并不少见。如何合理的通过Cache技术让系统性能得到较大的提升也不是通过寥寥几笔就能说明的清楚，这里我仅根据以往的经验列举一下什么样的数据适合通过Cache技术来提高系统性能：

1. 系统各种配置及规则数据；
   由于这些配置信息变动的频率非常低，访问概率又很高，所以非常适合存使用Cache；
2. 活跃用户的基本信息数据；
   虽然我们经常会听到某某网站的用户量达到成百上千万，但是很少有系统的活跃用户量能够都达到这个数量级。也很少有用户每天没事干去将自己的基本信息改来改去。更为重要的一点是用户的基本信息在应用系统中的访问频率极其频繁。所以用户基本信息的Cache，很容易让整个应用系统的性能出现一个质的提升。
3. 活跃用户的个性化定制信息数据；
   虽然用户个性化定制的数据从访问频率来看，可能并没有用户的基本信息那么的频繁，但相对于系统整体来说，也占了很大的比例，而且变更皮律一样不会太多。从Ebay的PayPal通过MySQL的Memory存储引擎实现用户个性化定制数据的成功案例我们就能看出对这部分信息进行Cache的价值了。虽然通过MySQL的Memory存储引擎并不像我们传统意义层面的Cache机制，但正是对Cache技术的合理利用和扩充造就了项目整体的成功。
4. 准实时的统计信息数据；
   所谓准实时的统计数据，实际上就是基于时间段的统计数据。这种数据不会实时更新，也很少需要增量更新，只有当达到重新Build该统计数据的时候需要做一次全量更新操作。虽然这种数据即使通过数据库来读取效率可能也会比较高，但是执行频率很高之后，同样会消耗不少资源。既然数据库服务器的资源非常珍贵，我们为什么不能放在应用相关的内存Cache中呢？
5. 其他一些访问频繁但变更较少的数据；
   出了上面这四种数据之外，在我们面对的各种系统环境中肯定还会有各种各样的变更较少但是访问很频繁的数据。只要合适，我们都可以将对他们的访问从数据库移到Cache中。

我们的数据层实现都是最精简的吗？

从以往的经验来看，一个合理的数据存取实现和一个拙劣的实现相比，在性能方面的差异经常会超出一个甚至几个数量级。我们先来分析一个非常简单且经常会遇到类似情况的示例：

在我们的示例网站系统中，现在要实现每个用户查看各自相册列表（假设每个列表显示10张相片）的时候，能够在相片名称后面显示该相片的留言数量。这个需求大家认为应该如何实现呢？我想90%的开发开发工程师会通过如下两步来实现该需求：

1、通过“SELECT id,subject,url FROM photo WHERE user_id = ? limit 10” 得到第一页的相片

相关信息；

2、通过第 1 步结果集中的 10 个相片 id 循环运行十次“SELECT COUNT(*) FROM photo_comment WHERE photh_id = ?” 来得到每张相册的回复数量然后再瓶装展现对象。

此外可能还有部分人想到了如下的方案：

1、和上面完全一样的操作步骤；

2、通过程序拼装上面得到的 10 个 photo 的 id，再通过 in 查询“SELECT photo_id,count(*) FROM photo_comment WHERE photo_id in (?) GROUP BY photo_id” 一次得到 10 个 photo 的所有回复数量，再组装两个结果集得到展现对象。

我们来对以上两个方案做一下简单的比较：

1、从 MySQL 执行的 SQL 数量来看 ，第一种解决方案为 11（1+10=11）条 SQL 语句，第二种解决方案为 2 条 SQL 语句（1+1）；

2、从应用程序与数据库交互来看，第一种为 11 次，第二种为 2 次；

3、从数据库的 IO 操作来看，简单假设每次 SQL 为 1 个 IO，第一种最少 11 次 IO，第二种小于等于 11 次 IO，而且只有当数据非常之离散的情况下才会需要 11 次；

4、从数据库处理的查询复杂度来看，第一种为两类很简单的查询，第二种有一条 SQL 语句有 GROUP BY 操作，比第一种解决方案增加了了排序分组操作；

5、从应用程序结果集处理来看，第一种 11 次结果集的处理，第二中 2 次结果集的处理，但是第二种解决方案中第二词结果处理数量是第一次的 10 倍；

6、从应用程序数据处理来看，第二种比第一种多了一个拼装 photo_id 的过程。

我们先从以上 6 点来做一个性能消耗的分析：

1、由于 MySQL 对客户端每次提交的 SQL 不管是相同还是不同，都需要进行完全解析，这个动作主要消耗的资源是数据库主机的 CPU，那么这里第一种方案和第二种方案消耗 CPU 的比例是 11:2。SQL 语句的解析动作在整个 SQL 语句执行过程中的整体消耗的 CPU 比例是较多的;

2、应用程序与数据库交互所消耗的资源基本上都在网络方面，同样也是 11：2；

3、数据库 IO 操作资源消耗为小于或者等于 1：1；

4、第二种解决方案需要比第一种多消耗内存资源进行排序分组操作，由于数据量不大，多出的消耗在语句整体消耗中占用比例会比较小，大概不会超过 20%，大家可以针对性测试；

5、结果集处理次数也为 11：2，但是第二中解决方案第二次处理数量较大，整体来说两次的性能消耗区别不大；

6、应用程序数据处理方面所多出的这个 photo_id 的拼装所消耗的资源是非常小的，甚至比应用程序与 MySQL 做一次简单的交互所消耗的资源还要少。

综合上面的这 6 点比较，我们可以很容易得出结论，从整体资源消耗来看，第二中方案会远远优于第一种解决方案。而在实际开发过程中，我们的程序员却很少选用。主要原因其实有两个，一个是第二种方案在程序代码实现方面可能会比第一种方案略为复杂，尤其是在当前编程环境中面向对象思想的普及，开发工程师可能会更习惯于以对象为中心的思考方式来解决问题。还有一个原因就是我们的程序员可能对 SQL 语句的使用并不是特别的熟悉，并不一定能够想到第二条 SQL 语句所实现的功能。对于第一个原因，我们可能只能通过加强开发工程师的性能优化意识来让大家能够自觉纠正，而第二个原因的解决就正是需要我们出马的时候了。 SQL 语句正是我们的专长，定期对开发工程师进行一些相应的数据库知识包括 SQL 语句方面的优化培训，可能会给大家带来意想不到的收获的。

这里我们还仅仅只是通过一个很长见的简单示例来说明数据层架构实现的区别对整体性能的影响，实际上可以简单的归结为过渡依赖嵌套循环的使用或者说是过渡弱化 SQL 语句的功能造成性能消耗过多的实例。后面我将进一步分析一下更多的因为架构实现差异所带来的性能消耗差异。

过度依赖数据库 SQL 语句的功能造成数据库操作效率低下

前面的案例是开发工程师过渡弱化 SQL 语句的功能造成的资源浪费案例，而这里我们再来分析一个完全相反的案例：在群组简介页面需要显示群名称和简介，每个群成员的 nick_name，以及群主的个人签名信息。

需求中所需信息存放在以下四个表中：user，user_profile，groups，user_group

我们先看看最简单的实现方法，一条 SQL 语句搞定所有事情：

```
SELECT name,description,user_type,nick_name,sign
FROM groups,user_group,user ,user_profile
WHERE groups.id = ?
    AND groups.id = user_group.group_id
    AND user_group.user_id = user.id
    AND user_profile.user_id = user.id
```

当然我们也可以通过如下稍微复杂一点的方法分两步搞定：

首先取得所有需要展示的 group 的相关信息和所有群组员的 nick_name 信息和组员类别：

```
SELECT name,description,user_type,nick_name
FROM groups,user_group,user
WHERE groups.id = ?
    AND groups.id = user_group.group_id
    AND user_group.user_id = user.id
```

然后在程序中通过上面结果集中的 user_type 找到群主的 user_id 再到 user_profile 表中取得群主的签名信息：

```
SELECT sign FROM user_profile WHERE user_id = ?
```

大家应该能够看出两者的区别吧，两种解决方案最大的区别在于交互次数和 SQL 复杂度。而带来的实际影响是第一种解决方案对 user_profile 表有不必要的访问（非群主的 profile 信息），造成 IO 访问的直接增加在 20%左右。而大家都知道，IO 操作在数据库应用系统中是非常昂贵的资源。尤其是当这个功能的 PV 较大的时候，第一种方案造成的 IO 损失是相当大的。

重复执行相同的 SQL 造成资源浪费

这个问题其实是每个人都非常清楚也完全认同的一个问题，但是在应用系统开发过程中，仍然会常有这样的现象存在。究其原因，主要还是开发工程师思维中面向对象的概念太过深入，以及为了减少自己代码开发的逻辑和对程序接口过度依赖所造成的。

我曾经在一个性能优化项目中遇到过一个案例，某个功能页面一侧是“分组”列表，是一列“分

组”的名字。页面主要内容则是该“分组”的所有“项目”列表。每个“项目”以名称（或者图标）显示，同时还有一个 SEO 相关的需求就是每个“项目”名称的链接地址中是需要有“分组”的名称的。所以在“项目”列表的每个 “项目”的展示内容中就需要得到该项目所属的组的名称。按照开发工程师开发思路，非常容易产生取得所有“项目”结果集并映射成相应对象之后，再从对象集中获取“项目”所属组的标识字段，然后循环到“分组”表中取得需要的”组名“。然后再将拼装成展示对象。

看到这里，我想大家应该已经知道这里存在的一个最大的问题就是多次重复执行了完全相同的 SQL 得到完全相同的内容。同时还犯了前面第一个案例中所犯的错误。或许大家看到之后会不相信有这样的案例存在，我可以非常肯定的告诉大家，事实就是这样。同时也请大家如果有条件的话，好好 Review 自己所在的系统的代码，非常有可能同样存在上面类似的情形。

还有部分解决方案要远优于上面的做法，那就是不循环去取了，而是通过 Join 一次完成，也就是解决了第一个案例所描述的性能问题。但是又误入了类似于第二个案例所描述的陷阱中了，因为实际上他只需要一次查询就可以得到所有“项目”所属的“分组”的名称（所有项目都是同一个组的）。

当然，也有部分解决方案也避免了第二个案例的问题，分为两条 SQL，两步完成了这个需求。这样在性能上面基本上也将近是数量级的提升了。

但是这就是性能最优的解决方案了么？不是的，我们甚至可以连一次都不需要访问就获得所需要的“分组”名称。首先，侧栏中的“分组”列表是需要有名称的，我们为什么不能直接利用到呢？

当然，可能有些系统的架构决定了侧栏和主要内容显示区来源于不同的模板（或者其他结构），那么我们也完全可以通过在进入这个功能页面的链接请求中通过参数传入我们需要的“分组”名称。这样我们就可以完全不需要根据“项目”相关信息去数据库获取所属“分组”的信息，就可以完成相应需求了。当然，是否需要通过请求参数来节省最后的这一次访问，可能会根据这个功能页面的 PV 来决定，如果访问并不是非常频繁，那么这个节省可能并不是很明显，而应用系统的复杂度却有所增加，而且程序看上去可能也会不够优雅，但是如果访问非常频繁的场景中，所节省的资源还是比较可观的。

上面还仅仅只是列举了我们平时比较常见的一些实现差异对性能所带来的影响，除了这些实现方面所带来的问题之外，应用系统的整体架构实现设计对系统性能的影响可能会更严重。下面大概列举了一些较为常见的架构设计实现不当带来的性能问题和资源浪费情况。

1、Cache 系统的不合理利用导致 Cache 命中率低下造成数据库访问量的增加，同时也浪费了 Cache 系统的硬件资源投入；

2、过度依赖面向对象思想，对系统

3、对可扩展性的过渡追求，促使系统设计的时候将对象拆得过于离散，造成系统中大量的复杂 Join 语句，而 MySQL Server 在各数据库系统中的主要优势在于处理简单逻辑的查询，这与其锁定的机制也有较大关系；

4、对数据库的过渡依赖，将大量更适合存放于文件系统中的数据存入了数据库中，造成数据库资源的浪费，影响到系统的整体性能，如各种日志信息；

5、过度理想化系统的用户体验，使大量非核心业务消耗过多的资源，如大量不需要实时更新的数据做了实时统计计算。

以上仅仅是一些比较常见的症结，在各种不同的应用环境中肯定还会有很多不同的性能问题，可能

需要大家通过仔细的数据分析和对系统的充分了解才能找到，但是一旦找到症结所在，通过相应的优化措施，所带来的收益也是相当可观的。

6.3 Query 语句对系统性能的影响

前面一节我们介绍了应用系统的实现差异对数据库应用系统整体性能的影响，这一节我们将分析SQL语句的差异对系统性能的影响。

我想对于各位读者来说，肯定都清楚 SQL 语句的优劣是对性能有影响的，但是到底有多大影响可能每个人都会有不同的体会，每个 SQL 语句在优化之前和优化之后的性能差异也是各不相同，所以对于性能差异到底有多大这个问题我们我们这里就不做详细分析了。我们重点分析实现同样功能的不同 SQL 语句在性能方面会产生较大的差异的根本原因，并通过一个较为典型的示例来对我们的分析做出相应的验证。

为什么返回完全相同结果集的不同SQL语句，在执行性能方面存在差异呢？这里我们先从SQL语句在数据库中执行并获取所需数据这个过程来做一个大概的分析了。

当 MySQL Server 的连接线程接收到 Client 端发送过来的 SQL 请求之后，会经过一系列的分解 Parse，进行相应的分析。然后，MySQL 会通过查询优化器模块（Optimizer）根据该 SQL 所设涉及到的数据表的相关统计信息进行计算分析，然后再得出一个 MySQL 认为最合理最优化的数据访问方式，也就是我们常说的“执行计划”，然后再根据所得到的执行计划通过调用存储引擎借口来获取相应数据。然后再将存储引擎返回的数据进行相关处理，并以 Client 端所要求的格式作为结果集返回给 Client 端的应用程序。

注：这里所说的统计数据，是我们通过ANALYZE TABLE命令通知MySQL对表的相关数据做分析之后所获得到的一些数据统计量。这些统计数据对 MySQL 优化器而言是非常重要的，优化器所生成的执行计划的好坏，主要就是由这些统计数据所决定的。实际上，在其他一些数据库管理软件中也有类似相应的统计数据。

我们都知道，在数据库管理软件中，最大的性能瓶颈就是在于磁盘 IO，也就是数据的存取操作上面。而对于同一份数据，当我们以不同方式去寻找其中的某一点内容的时候，所需要读取的数据量可能会有天壤之别，所消耗的资源也自然是区别甚大。所以，当我们需要从数据库中查询某个数据的时候，所消耗资源的多少主要就取决于数据库以一个什么样的数据读取方式来完成我们的查询请求，也就是取决于 SQL 语句的执行计划。

对于唯一一个SQL语句来说，经过MySQL Parse之后分解的结构都是固定的，只要统计信息稳定，其执行计划基本上都是比较固定的。而不同写法的SQL语句，经过MySQL Parse之后分解的结构结构就可能完全不同，即使优化器使用完全一样的统计信息来进行优化，最后所得出的执行计划也可能完全不一样。而执行计划又是决定一个SQL语句最终的资源消耗量的主要因素。所以，实现功能完全一样的SQL语句，在性能上面可能会有差别巨大的性能消耗。当然，如果功能一样，而且经过 MySQL 的优化器优化之后的执行计划也完全一致的不同 SQL 语句在资源消耗方面可能就相差很小了。当然这里所指的消耗主要是 IO 资源的消耗，并不包括 CPU 的消耗。

下面我们将通过一两个具体的示例来分析写法不一样而功能完全相同的两条 SQL 的在性能方面的差异。

示例一

需求：取出某个 group（假设 id 为 100）下的用户编号（id），用户昵称（nick_name）、用户性别（sexuality）、用户签名（sign）和用户生日（birthday），并按照加入组的时间（user_group.gmt_create）来进行倒序排列，取出前 20 个。

解决方案一、

```
SELECT id,nick_name
FROM user,user_group
WHERE user_group.group_id = 1
    and user_group.user_id = user.id
limit 100,20;
```

解决方案二、

```
SELECT user.id,user.nick_name
FROM (
    SELECT user_id
    FROM user_group
    WHERE user_group.group_id = 1
    ORDER BY gmt_create desc
    limit 100,20) t,user
WHERE t.user_id = user.id;
```

我们先来看看执行计划：

```
sky@localhost : example 10:32:13> explain
    -> SELECT id,nick_name
    ->     FROM user,user_group
    ->     WHERE user_group.group_id = 1
    ->         and user_group.user_id = user.id
    ->     ORDER BY user_group.gmt_create desc
    ->     limit 100,20\G
    *************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user_group
         type: ref
possible_keys: user_group_uid_gid_ind,user_group_gid_ind
          key: user_group_gid_ind
      key_len: 4
          ref: const
         rows: 31156
```

```
        Extra: Using where; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: example.user_group.user_id
         rows: 1
        Extra:

sky@localhost : example 10:32:20> explain
    ->  SELECT user.id,user.nick_name
    ->     FROM (
    ->         SELECT user_id
    ->         FROM user_group
    ->         WHERE user_group.group_id = 1
    ->         ORDER BY gmt_create desc
    ->         limit 100,20) t,user
    ->     WHERE t.user_id = user.id\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 20
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: user
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: t.user_id
         rows: 1
        Extra:
```

```
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: user_group
         type: ref
possible_keys: user_group_gid_ind
          key: user_group_gid_ind
      key_len: 4
          ref: const
         rows: 31156
        Extra: Using filesort
```

执行计划对比分析：

解决方案一中的执行计划显示MySQL在对两个参与Join的表都利用到了索引，user_group表利用了user_group_gid_ind索引（key: user_group_gid_ind），user表利用到了主键索引（key: PRIMARY），在参与Join前MySQL通过Where过滤后的结果集与user表进行Join，最后通过排序取出Join后结果的“limit 100,20”条结果返回。

解决方案二的SQL语句利用到了子查询，所以执行计划会稍微复杂一些，首先可以看到两个表都和解决方案1一样都利用到了索引（所使用的索引也完全一样），执行计划显示该子查询以user_group为驱动，也就是先通过user_group进行过滤并马上进行这一论的结果集排序，也就取得了SQL中的“limit 100,20”条结果，然后与user表进行Join，得到相应的数据。这里可能有人会怀疑在自查询中从user_group表所取得与user表参与 Join的记录条数并不是20条，而是整个group_id=1的所有结果。那么清大家看看该执行计划中的第一行，该行内容就充分说明了在外层查询中的所有的20条记录全部被返回。

通过比较两个解决方案的执行计划，我们可以看到第一中解决方案中需要和user表参与Join的记录数MySQL通过统计数据估算出来是31156，也就是通过user_group表返回的所有满足group_id=1的记录数（系统中的实际数据是20000）。而第二种解决方案的执行计划中，user 表参与Join的数据就只有20条，两者相差很大，通过本节最初的分析，我们认为第二中解决方案应该明显优于第一种解决方案。

下面我们通过对比两个解决觉方案的SQL实际执行的profile详细信息，来验证我们上面的判断。由于SQL语句执行所消耗的最大两部分资源就是IO和CPU，所以这里为了节约篇幅，仅列出BLOCK IO和CPU两项profile信息（Query Profiler 的详细介绍将在后面章节中独立介绍）：

先打开profiling功能，然后分别执行两个解决方案的SQL语句：

```
sky@localhost : example 10:46:43> set profiling = 1;
Query OK, 0 rows affected (0.00 sec)

sky@localhost : example 10:46:50> SELECT id,nick_name
    ->      FROM user,user_group
    ->      WHERE user_group.group_id = 1
    ->          and user_group.user_id = user.id
    ->      ORDER BY user_group.gmt_create desc
```

```
    ->     limit 100,20;
+--------+-----------+
| id     | nick_name |
+--------+-----------+
| 990101 | 990101    |
| 990102 | 990102    |
| 990103 | 990103    |
| 990104 | 990104    |
| 990105 | 990105    |
| 990106 | 990106    |
| 990107 | 990107    |
| 990108 | 990108    |
| 990109 | 990109    |
| 990110 | 990110    |
| 990111 | 990111    |
| 990112 | 990112    |
| 990113 | 990113    |
| 990114 | 990114    |
| 990115 | 990115    |
| 990116 | 990116    |
| 990117 | 990117    |
| 990118 | 990118    |
| 990119 | 990119    |
| 990120 | 990120    |
+--------+-----------+
20 rows in set (1.02 sec)

sky@localhost : example 10:46:58> SELECT user.id,user.nick_name
    ->     FROM (
    ->         SELECT user_id
    ->         FROM user_group
    ->         WHERE user_group.group_id = 1
    ->         ORDER BY gmt_create desc
    ->         limit 100,20) t,user
    ->     WHERE t.user_id = user.id;
+--------+-----------+
| id     | nick_name |
+--------+-----------+
| 990101 | 990101    |
| 990102 | 990102    |
| 990103 | 990103    |
| 990104 | 990104    |
| 990105 | 990105    |
| 990106 | 990106    |
```

```
| 990107 | 990107    |
| 990108 | 990108    |
| 990109 | 990109    |
| 990110 | 990110    |
| 990111 | 990111    |
| 990112 | 990112    |
| 990113 | 990113    |
| 990114 | 990114    |
| 990115 | 990115    |
| 990116 | 990116    |
| 990117 | 990117    |
| 990118 | 990118    |
| 990119 | 990119    |
| 990120 | 990120    |
+--------+-----------+
20 rows in set (0.96 sec)
```

查看系统中的 profile 信息，刚刚执行的两个 SQL 语句的执行 profile 信息已经记录下来了：

```
sky@localhost : example 10:47:07> show profiles\G
*************************** 1. row ***************************
Query_ID: 1
Duration: 1.02367600
   Query: SELECT id,nick_name
    FROM user,user_group
    WHERE user_group.group_id = 1
        and user_group.user_id = user.id
    ORDER BY user_group.gmt_create desc
    limit 100,20
*************************** 2. row ***************************
Query_ID: 2
Duration: 0.96327800
   Query: SELECT user.id,user.nick_name
    FROM (
        SELECT user_id
        FROM user_group
        WHERE user_group.group_id = 1
        ORDER BY gmt_create desc
        limit 100,20) t,user
    WHERE t.user_id = user.id
2 rows in set (0.00 sec)

sky@localhost : example 10:47:34> SHOW profile CPU,BLOCK IO io FOR query  1;
```

```
+--------------------+----------+-----------+------------+--------------+---------------+
| Status             | Duration | CPU_user  | CPU_system | Block_ops_in | Block_ops_out |
+--------------------+----------+-----------+------------+--------------+---------------+
| (initialization)   | 0.000068 | 0         | 0          |            0 |             0 |
| Opening tables     | 0.000015 | 0         | 0          |            0 |             0 |
| System lock        | 0.000006 | 0         | 0          |            0 |             0 |
| Table lock         | 0.000009 | 0         | 0          |            0 |             0 |
| init               | 0.000026 | 0         | 0          |            0 |             0 |
| optimizing         | 0.000014 | 0         | 0          |            0 |             0 |
| statistics         | 0.000068 | 0         | 0          |            0 |             0 |
| preparing          | 0.000019 | 0         | 0          |            0 |             0 |
| executing          | 0.000004 | 0         | 0          |            0 |             0 |
| Sorting result     | 1.03614  | 0.5600349 | 0.428027   |            0 |         15632 |
| Sending data       | 0.071047 | 0         | 0.004      |           88 |             0 |
| end                | 0.000012 | 0         | 0          |            0 |             0 |
| query end          | 0.000006 | 0         | 0          |            0 |             0 |
| freeing items      | 0.000012 | 0         | 0          |            0 |             0 |
| closing tables     | 0.000007 | 0         | 0          |            0 |             0 |
| logging slow query | 0.000003 | 0         | 0          |            0 |             0 |
+--------------------+----------+-----------+------------+--------------+---------------+
16 rows in set (0.00 sec)

sky@localhost : example 10:47:40> SHOW profile CPU,BLOCK IO io FOR query 2;
+--------------------+----------+----------+------------+--------------+---------------+
| Status             | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+--------------------+----------+----------+------------+--------------+---------------+
| (initialization)   | 0.000087 | 0        | 0          |            0 |             0 |
| Opening tables     | 0.000018 | 0        | 0          |            0 |             0 |
| System lock        | 0.000007 | 0        | 0          |            0 |             0 |
| Table lock         | 0.000059 | 0        | 0          |            0 |             0 |
| optimizing         | 0.00001  | 0        | 0          |            0 |             0 |
| statistics         | 0.000068 | 0        | 0          |            0 |             0 |
| preparing          | 0.000017 | 0        | 0          |            0 |             0 |
| executing          | 0.000004 | 0        | 0          |            0 |             0 |
| Sorting result     | 0.928184 | 0.572035 | 0.352022   |            0 |            32 |
| Sending data       | 0.000112 | 0        | 0          |            0 |             0 |
| init               | 0.000025 | 0        | 0          |            0 |             0 |
| optimizing         | 0.000012 | 0        | 0          |            0 |             0 |
| statistics         | 0.000025 | 0        | 0          |            0 |             0 |
| preparing          | 0.000013 | 0        | 0          |            0 |             0 |
| executing          | 0.000004 | 0        | 0          |            0 |             0 |
| Sending data       | 0.000241 | 0        | 0          |            0 |             0 |
| end                | 0.000005 | 0        | 0          |            0 |             0 |
| query end          | 0.000006 | 0        | 0          |            0 |             0 |
```

```
| freeing items      | 0.000015 | 0        | 0          |            0 |             0 |
| closing tables     | 0.000004 | 0        | 0          |            0 |             0 |
| removing tmp table | 0.000019 | 0        | 0          |            0 |             0 |
| closing tables     | 0.000005 | 0        | 0          |            0 |             0 |
| logging slow query | 0.000004 | 0        | 0          |            0 |             0 |
+--------------------+----------+----------+------------+--------------+---------------+
```

我们先看看两条 SQL 执行中的 IO 消耗，两者区别就在于“Sorting result”，我们回顾一下前面执行计划的对比，两个解决方案的排序过滤数据的时机不一样，排序后需要取得的数据量一个是 20000，一个是 20，正好和这里的 profile 信息吻合，第一种解决方案的“Sorting result”的 IO 值是第二种解决方案的将近 500 倍。

然后再来看看 CPU 消耗，所有消耗中，消耗最大的也是“Sorting result”这一项，第一个消耗多出的缘由和上面 IO 消耗差异是一样的。

结论：

通过上面两条功能完全相同的 SQL 语句的执行计划分析，以及通过实际执行后的 profile 数据的验证，都证明了第二种解决方案优于第一种解决方案。同时通过后者的实际验证，也再次证明了我们前面所做的执行计划基本决定了 SQL 语句性能。

6.4 Schema 设计对系统的性能影响

前面两节中，我们已经分析了在一个数据库应用系统的软环境中应用系统的架构实现和系统中与数据库交互的 SQL 语句对系统性能的影响。在这一节我们再分析一下系统的数据模型设计实现对系统的性能影响，更通俗一点就是数据库的 Schema 设计对系统性能的影响。

在很多人看来，数据库 Schema 设计是一件非常简单的事情，就大体按照系统设计时候的相关实体对象对应成一个一个的表格基本上就可以了。然后为了在功能上做到尽可能容易扩展，再根据数据库范式规则进行调整，做到第三范式或者第四范式，基本就算完事了。

数据库 Schema 设计真的有如上面所说的这么简单么？可以非常肯定的告诉大家，数据库 Schema 设计所需要做的事情远远不止如此。如果您之前的数据库 Schema 设计一直都是这么做的，那么在该设计应用于正式环境之后，很可能带来非常大的性能代价。

由于在后面的“MySQL 数据库应用系统设计”中的“系统架构最优化“这一节中会介较为详细的从性能优化的角度来分析如何如何设计数据库 Schema，所以这里暂时先不介绍如何来设计性能优异的数据库 Schema 结构，仅仅通过一个实际的示例来展示 Schema 结构的不一样在性能方面所带来的差异。

需求概述：一个简单的讨论区系统，需要有用户，用户组，组讨论区这三部分基本功能

简要分析：1、需要存放用户数据的表；

2、需要存放分组信息和存放用户与组关系的表

3、需要存放讨论信息的表；

解决方案：

原始方案一：分别用用四个表来存放用户，分组，用户与组关系以及各组的讨论帖子的信息如下：

user 用户表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		0	
nick_name	varchar(32)	NO		NULL	
password	char(64)	YES		NULL	
email	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	
sexuality	char(1)	NO		NULL	
msn	varchar(32)	YES		NULL	
sign	varchar(64)	YES		NULL	
birthday	date	YES		NULL	
hobby	varchar(64)	YES		NULL	
location	varchar(64)	YES		NULL	
description	varchar(1024)	YES		NULL	

groups 分组表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
name	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	
description	varchar(1024)	YES		NULL	

user_group 关系表：

Field	Type	Null	Key	Default	Extra
user_id	int(11)	NO	MUL	NULL	
group_id	int(11)	NO	MUL	NULL	
user_type	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
status	varchar(16)	NO		NULL	

group_message 讨论组帖子表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
group_id	int(11)	NO		NULL	
user_id	int(11)	NO		NULL	
subject	varchar(128)	NO		NULL	
content	text	YES		NULL	

优化后方案二：

user 用户表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		0	
nick_name	varchar(32)	NO		NULL	
password	char(64)	YES		NULL	
email	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	

user_profile 用户属性表（记录与 user 一一对应）：

Field	Type	Null	Key	Default	Extra
sexuality	char(1)	NO		NULL	
msn	varchar(32)	YES		NULL	
sign	varchar(64)	YES		NULL	
birthday	date	YES		NULL	
hobby	varchar(64)	YES		NULL	
location	varchar(64)	YES		NULL	
description	varchar(1024)	YES		NULL	

groups 和 user_group 这两个表和方案一完全一样

group_message 讨论组帖子表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
group_id	int(11)	NO		NULL	
user_id	int(11)	NO		NULL	
author	varchar(32)	NO		NULL	
subject	varchar(128)	NO		NULL	

group_message_content 帖子内容表（记录与 group_message 一一对应）：

Field	Type	Null	Key	Default	Extra
group_msg_id	int(11)	NO		NULL	
content	text	NO		NULL	

我们先来比较一下两个解决方案所设计的 Schema 的区别。区别主要体现在两点，一个区别是在 group_message 表中增加了 author 字段来存放发帖作者的昵称，与 user 表的 nick_name 相对应，另外一个就是第二个解决方案将 user 表和 group_message 表都分拆成了两个表，关系分别都是一一对应。

方案二看上去比方案一要更复杂一些，首先是表的数量多了 2 个，然后是在 group_message 中冗余存放了作者昵称。我们试想一下，一个讨论区系统，访问最多的页面会是什么？我想大家都会很清楚是帖子标题列表页面。而帖子标题列表页面最主要的信息就是都是来自 group_message 表中，同时帖子标题后面的作者一般都是通过用户名成（昵称）来展示。按照第一种解决方案来设计的 Schema，我们就需要执行类似如下这样的 SQL 语句来得到数据：

```
SELECT t.id, t.subject,user.id, u.nick_name
FROM (
    SELECT id, user_id, subject
    FROM group_message
    WHERE group_id = ?
    ORDER BY gmt_modified DESC LIMIT 20
) t, user u
WHERE t.user_id = u.id
```

但是第二中解决方案所需要执行的 SQL 就会简单很多，如下：

```
SELECT t.id, t.subject, t.user_id, t.author
FROM group_message
WHERE group_id = ?
ORDER BY gmt_modified DESC LIMIT 20
```

两个 SQL 相比较，大家都能很明显的看出谁优谁劣了，第一个是需要读取两个表的数据进行 Join，

与第二个 SQL 相比性能差距很大，尤其是如果第一个再写的差一点，性能更是非常糟糕，两者所带来的资源消耗就更相差玄虚了。

不仅仅如此，由于第一个方案中的 group_message 表中还包含一个大字段“content”，该字段所存放的信息要占整个表的绝大部分存储空间，但在这条系统中执行最频繁的 SQL 之一中是完全不需要该字段所存放信息的，但是由于这个 SQL 又没办法做到不访问 group_message 表的数据，所以第一条 SQL 在数据读取过程中会需要读取大量没有任何意义的数据。

在系统中用户数据的读取也是比较频繁的，但是大多数地方所需要的用户数据都只是用户的几个基本属性，如用户的 id，昵称，密码，状态，邮箱等，所以将用户表的这几个属性单独分离出来后，也会让大量的 SQL 语句在运行的时候减少数据的检索量，从而提高性能。

可能有人会觉得，在我们将一个表分成两个表的时候，我们如果要访问被分拆出去的信息的时候，性能不是就会变差了吗？是的，对于那些需要访问如 user 的 sign，msn 等原来只需要一个表就可以完成的 SQL 来说，现在都需要两条 SQL 来完成，性能确实会 有所降低，但是由于两个表都是一对一的关联关系，关联字段的过滤性也非常高，而且这样的查询需求在整个系统中所占有的比例也并不高，所以这里所带来的性能损失实际上要远远小于在其他 SQL 上所节省出来的资源，所以完全不必为此担心

6.5 硬件环境对系统性能的影响

在本章之前的所有部分都是介绍的整个系统中的软件环境对系统性能的影响，这一节我们将从系统硬件环境来分析对数据库系统的影响，并从数据库服务器主机的角度来做一些针对性的优化建议。

任何一个系统的硬件环境都会对起性能起到非常关键的作用，这一点我想每一位读者朋友都是非常清楚的。而数据库应用系统环境中，由于数据库自身的特点和在系统中的角色决定了他在整个系统中是最难以扩展的部分。所以在大多数环境下，数据库服务器主机（或者主机集群）的性能在很大程度上决定了整个应用系统的性能。

既然我们的数据库主机资源如此重要，肯定很多读者朋友会希望知道，数据库服务器主机的各部分硬件到底谁最重要，各部分对整体性能的影响各自所占的比例是多少，以便能够根据这些比例选取合适的主机机型作为数据库主机。但是我只能很遗憾的告诉大家，没有任何一个定律或者法则可以很准确的给出这个答案。

当然，大家也不必太沮丧。虽然没有哪个法则可以准确的知道我们到底该如何选配一个主机的各部分硬件，但是根据应用类型的不同，总体上还是有一个可以大致遵循的原则可以参考的。

首先，数据库主机是存取数据的地方，那么其 IO 操作自然不会少，所以数据库主机的 IO 性能肯定是需要最优先考虑的一个因素，这一点不管是什么类型的数据库应用都是适用的。不过，这里的 IO 性能并不仅仅只是指物理的磁盘 IO，而是主机的整体 IO 性能，是主机整个 IO 系统的总体 IO 性能。而 IO 性能本身又可以分为两类，一类是每秒可提供的 IO 访问次数，也就是我们常说的 IOPS 数量，还有一种就是每秒的 IO 总流量，也就是我们常说的 IO 吞吐量。在主机中决定 IO 性能部件主要由磁盘和内存所决定，当然也包括各种与 IO 相关的板卡。

其次，由于数据库主机和普通的应用程序服务器相比，资源要相对集中很多，单台主机上所需要进行的计算量自然也就比较多，所以数据库主机的 CPU 处理能力也不能忽视。

最后，由于数据库负责数据的存储，与各应用程序的交互中传递的数据量比其他各类服务器都要多，所以数据库主机的网络设备的性能也可能会成为系统的瓶颈。

由于上面这三类部件是影响数据库主机性能的最主要因素，其他部件成为性能瓶颈的几率要小很多，所以后面我们通过对各种类型的应用做一个简单的分析，再针对性的给出这三类部件的基本选型建议。

1、典型 OLTP 应用系统

对于各种数据库系统环境中大家最常见的 OLTP 系统，其特点是并发量大，整体数据量比较多，但每次访问的数据比较少，且访问的数据比较离散，活跃数据占总体数据的比例不是太大。对于这类系统的数据库实际上是最难维护，最难以优化的，对主机整体性能要求也是最高的。因为他不仅访问量很高，数据量也不小。

针对上面的这些特点和分析，我们可以对 OLTP 的得出一个大致的方向。

虽然系统总体数据量较大，但是系统活跃数据在数据总量中所占的比例不大，那么我们可以通过扩大内存容量来尽可能多的将活跃数据 cache 到内存中；

虽然 IO 访问非常频繁，但是每次访问的数据量较少且很离散，那么我们对磁盘存储的要求是 IOPS 表现要很好，吞吐量是次要因素；

并发量很高，CPU 每秒所要处理的请求自然也就很多，所以 CPU 处理能力需要比较强劲；

虽然与客户端的每次交互的数据量并不是特别大，但是网络交互非常频繁，所以主机与客户端交互的网络设备对流量能力也要求不能太弱。

2、典型 OLAP 应用系统

用于数据分析的 OLAP 系统的主要特点就是数据量非常大，并发访问不多，但每次访问所需要检索的数据量都比较多，而且数据访问相对较为集中，没有太明显的活跃数据概念。

基于 OLAP 系统的各种特点和相应的分析，针对 OLAP 系统硬件优化的大致策略如下：

数据量非常大，所以磁盘存储系统的单位容量需要尽量大一些；

单次访问数据量较大，而且访问数据比较集中，那么对 IO 系统的性能要求是需要有尽可能大的每秒 IO 吞吐量，所以应该选用每秒吞吐量尽可能大的磁盘；

虽然 IO 性能要求也比较高，但是并发请求较少，所以 CPU 处理能力较难成为性能瓶颈，所以 CPU 处理能力没有太苛刻的要求；

虽然每次请求的访问量很大，但是执行过程中的数据大都不会返回给客户端，最终返回给客户端的数据量都较小，所以和客户端交互的网络设备要求并不是太高；

此外，由于 OLAP 系统由于其每次运算过程较长，可以很好的并行化，所以一般的 OLAP 系统都是由多台主机构成的一个集群，而集群中主机与主机之间的数据交互量一般来说都是非常大的，所以在集群中主机之间的网络设备要求很高。

3、除了以上两个典型应用之外，还有一类比较特殊的应用系统，他们的数据量不是特别大，但是访问请求及其频繁，而且大部分是读请求。可能每秒需要提供上万甚至几万次请求，每次请求都非常简

单，可能大部分都只有一条或者几条比较小的记录返回，就比如基于数据库的 DNS 服务就是这样类型的服务。

虽然数据量小，但是访问极其频繁，所以可以通过较大的内存来 cache 住大部分的数据，这能够保证非常高的命中率，磁盘 IO 量比较小，所以磁盘也不需要特别高性能的；

并发请求非常频繁，比需要较强的 CPU 处理能力才能处理；

虽然应用与数据库交互量非常大，但是每次交互数据较少，总体流量虽然也会较大，但是一般来说普通的千兆网卡已经足够了。

在很多人看来，性能的根本决定因素是硬件性能的好坏。但实际上，硬件性能只能在某些阶段对系统性能产生根本性影响。当我们的 CPU 处理能力足够的多，IO 系统的处理能力足够强的时候，如果我们的应用架构和业务实现不够优化，一个本来很简单的实现非得绕很多个弯子来回交互多次，那再强的硬件也没有用，因为来回的交互总是需要消耗时间。尤其是有些业务逻辑设计不是特别合理的应用，数据库 Schema 设计的不够合理，一个任务在系统中又被分拆成很多个步骤，每个步骤都使用了非常复杂的 Query 语句。笔者曾经就遇到过这样一个系统，该系统是购买的某知名厂商的一个项目管理软件。该系统最初运行在一台 Dell2950 的 PC Server 上面，使用者一直抱怨系统响应很慢，但我从服务器上面的状态来看系统并繁忙（系统并发不是太大）。后来使用者强烈要求通过更换硬件设施来提升系统性能，虽然我一直反对，但最后在管理层的要求下，更换成了一台 Sun 的 S880 小型机，主机 CPU 的处理能力至少是原来机器的 3 倍以上，存储系统也从原来使用本地磁盘换成使用 EMC 的中断存储 CX300。可在试用阶段，发现系统整体性能没有任何的提升，最终还是取消了更换硬件的计划。

所以，在应用系统的硬件配置方面，我们应该要以一个理性的眼光来看待，只有合适的才是最好的。并不是说硬件资源越好，系统性能就一定会越好。而且，硬件系统本身总是有一个扩展极限的，如果我们一味的希望通过升级硬件性能来解决系统的性能问题，那么总有一天将会遇到无法逾越的瓶颈。到那时候，就算有再多的钱去砸也无济于事了。

6.6 小结

虽然本章是以影响 MySQL Server 性能的相关因素来展开分析，但实际上很多内容都对于大多数数据库应用系统适用。数据库管理软件仅仅是实现了数据库应用系统中的数据存取操作，和数据的持久化。数据库应用系统的优化真正能带来最大收益的就是商业需求和系统架构及业务实现的优化，然后是数据库 Schema 设计的优化，然后才是 Query 语句的优化，最后才是数据库管理软件自身的一些优化。通过笔者的经验，在整个系统的性能优化中，如果按照百分比来划分上面几个层面的优化带来的性能收益，可以得出大概如下的数据：

需求和架构及业务实现优化：55%

Query 语句的优化：30%

数据库自身的优化：15%

很多时候，大家看到数据库应用系统中性能瓶颈出现在数据库方面，就希望通过数据库的优化来解决问题，但不管 DBA 对数据库多们了解，对 Query 语句的优化多么精通，最终还是很难解决整个系统的性能问题。原因就在于并没有真正找到根本的症结所在。

所以，数据库应用系统的优化，实际上是一个需要多方面配合，多方面优化的才能产生根本性改善的事情。简单来说，可以通过下面三句话来简单的概括数据库应用系统的性能优化：商业需求合理化，系统架构最优化，逻辑实现精简化，硬件设施理性化。

第 7 章 MySQL 数据库锁定机制

前言：

为了保证数据的一致完整性，任何一个数据库都存在锁定机制。锁定机制的优劣直接应想到一个数据库系统的并发处理能力和性能，所以锁定机制的实现也就成为了各种数据库的核心技术之一。本章将对 MySQL 中两种使用最为频繁的存储引擎 MyISAM 和 Innodb 各自的锁定机制进行较为详细的分析。

7.1 MySQL 锁定机制简介

数据库锁定机制简单来说就是数据库为了保证数据的一致性而使各种共享资源在被并发访问访问变得有序所设计的一种规则。对于任何一种数据库来说都需要有相应的锁定机制，所以 MySQL 自然也不能例外。MySQL 数据库由于其自身架构的特点，存在多种数据存储引擎，每种存储引擎所针对的应用场景特点都不太一样，为了满足各自特定应用场景的需求，每种存储引擎的锁定机制都是为各自所面对的特定场景而优化设计，所以各存储引擎的锁定机制也有较大区别。

总的来说，MySQL 各存储引擎使用了三种类型（级别）的锁定机制：行级锁定，页级锁定和表级锁定。下面我们先分析一下 MySQL 这三种锁定的特点和各自的优劣所在。

● 行级锁定（row-level）

行级锁定最大的特点就是锁定对象的颗粒度很小，也是目前各大数据库管理软件所实现的锁定颗粒度最小的。由于锁定颗粒度很小，所以发生锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。

虽然能够在并发处理能力上面有较大的优势，但是行级锁定也因此带来了不少弊端。由于锁定资源的颗粒度很小，所以每次获取锁和释放锁需要做的事情也更多，带来的消耗自然也就更大了。此外，行级锁定也最容易发生死锁。

● 表级锁定（table-level）

和行级锁定相反，表级别的锁定是 MySQL 各存储引擎中最大颗粒度的锁定机制。该锁定机制最大的

特点是实现逻辑非常简单，带来的系统负面影响最小。所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，所以可以很好的避免困扰我们的死锁问题。

当然，锁定颗粒度大所带来最大的负面影响就是出现锁定资源争用的概率也会最高，致使并大度大打折扣。

● 页级锁定（page-level）

页级锁定是 MySQL 中比较独特的一种锁定级别，在其他数据库管理软件中也并不是太常见。页级锁定的特点是锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样是介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。

在数据库实现资源锁定的过程中，随着锁定资源颗粒度的减小，锁定相同数据量的数据所需要消耗的内存数量是越来越多的，实现算法也会越来越复杂。不过，随着锁定资源颗粒度的减小，应用程序的访问请求遇到锁等待的可能性也会随之降低，系统整体并发度也随之提升。

在 MySQL 数据库中，使用表级锁定的主要是 MyISAM，Memory，CSV 等一些非事务性存储引擎，而使用行级锁定的主要是 Innodb 存储引擎和 NDB Cluster 存储引擎，页级锁定主要是 BerkeleyDB 存储引擎的锁定方式。

MySQL 的如此的锁定机制主要是由于其最初的历史所决定的。在最初，MySQL 希望设计一种完全独立于各种存储引擎的锁定机制，而且在早期的 MySQL 数据库中，MySQL 的存储引擎（MyISAM 和 Momery）的设计是建立在“任何表在同一时刻都只允许单个线程对其访问（包括读）”这样的假设之上。但是，随着 MySQL 的不断完善，系统的不断改进，在 MySQL3.23 版本开发的时候，MySQL 开发人员不得不修正之前的假设。因为他们发现一个线程正在读某个表的时候，另一个线程是可以对该表进行 insert 操作的，只不过只能 INSERT 到数据文件的最尾部。这也就是从 MySQL 从 3.23 版本开始提供的我们所说的 Concurrent Insert。

当出现 Concurrent Insert 之后，MySQL 的开发人员不得不修改之前系统中的锁定实现功能，但是仅仅只是增加了对 Concurrent Insert 的支持，并没有改动整体架构。可是在不久之后，随着 BerkeleyDB 存储引擎的引入，之前的锁定机制遇到了更大的挑战。因为 BerkeleyDB 存储引擎并没有 MyISAM 和 Memory 存储引擎同一时刻只允许单一线程访问某一个表的限制，而是将这个单线程访问限制的颗粒度缩小到了单个 page，这又一次迫使 MySQL 开发人员不得不再一次修改锁定机制的实现。

由于新的存储引擎的引入，导致锁定机制不能满足要求，让 MySQL 的人意识到已经不可能实现一种完全独立的满足各种存储引擎要求的锁定实现机制。如果因为锁定机制的拙劣实现而导致存储引擎的整体性能的下降，肯定会严重打击存储引擎提供者的积极性，这是 MySQL 公司非常不愿意看到的，因为这完全不符合 MySQL 的战略发展思路。所以工程师们不得不放弃了最初的设计初衷，在锁定实现机制中作出修改，允许存储引擎自己改变 MySQL 通过接口传入的锁定类型而自行决定该怎样锁定数据。

7．2 各种锁定机制分析

在整体了解了MySQL锁定机制之后，这一节我们将详细分析MySQL自身提供的表锁定机制和其他储引擎实自身实现的行锁定机制，并通过MyISAM存储引擎和Innodb存储引擎实例演示。

表级锁定

MySQL的表级锁定主要分为两种类型，一种是读锁定，另一种是写锁定。在MySQL中，主要通过四个队列来维护这两种锁定：两个存放当前正在锁定中的读和写锁定信息，另外两个存放等待中的读写锁定信息，如下：

- Current read-lock queue (lock->read)
- Pending read-lock queue (lock->read_wait)
- Current write-lock queue (lock->write)
- Pending write-lock queue (lock->write_wait)

当前持有读锁的所有线程的相关信息都能够在Current read-lock queue中找到，队列中的信息按照获取到锁的时间依序存放。而正在等待锁定资源的信息则存放在Pending read-lock queue 里面，另外两个存放写锁信息的队列也按照上面相同规则来存放信息。

虽然对于我们这些使用者来说MySQL展现出来的锁定（表锁定）只有读锁定和写锁定这两种类型，但是在MySQL内部实现中却有多达11种锁定类型，由系统中一个枚举量（thr_lock_type）定义，各值描述如下：

锁定类型	说明
IGNORE	当发生锁请求的时候内部交互使用，在锁定结构和队列中并不会有任何信息存储
UNLOCK	释放锁定请求的交互用所类型
READ	普通读锁定
WRITE	普通写锁定
READ_WITH_SHARED_LOCKS	在Innodb中使用到，由如下方式产生 如：SELECT ... LOCK IN SHARE MODE
READ_HIGH_PRIORITY	高优先级读锁定
READ_NO_INSERT	不允许Concurent Insert的锁定
WRITE_ALLOW_WRITE	这个类型实际上就是当由存储引擎自行处理锁定的时候，mysqld允许其他的线程再获取读或者写锁定，因为即使资源冲突，存储引擎自己也会知道怎么来处理
WRITE_ALLOW_READ	这种锁定发生在对表做DDL（ALTER TABLE ...）的时候，MySQL可以允许其他线程获取读锁定，因为MySQL是通过重建整个表然后再RENAME而实现的该功能，所在整个过程原表仍然可以提供读服务
WRITE_CONCURRENT_INSERT	正在进行Concurent Insert时候所使用的锁定方式，该锁定进行的时候，除了READ_NO_INSERT之外的其他任何读锁定请求都不会被阻塞
WRITE_DELAYED	在使用INSERT DELAYED时候的锁定类型
WRITE_LOW_PRIORITY	显示声明的低级别锁定方式，通过设置LOW_PRIORITY_UPDAT = 1 而产生
WRITE_ONLY	当在操作过程中某个锁定异常中断之后系统内部需要进行CLOSE TABLE操作，在这个过程中出现的锁定类型就是WRITE_ONLY

读锁定

一个新的客户端请求在申请获取读锁定资源的时候，需要满足两个条件：

1、 请求锁定的资源当前没有被写锁定；

2、 写锁定等待队列（Pending write-lock queue）中没有更高优先级的写锁定等待；

如果满足了上面两个条件之后，该请求会被立即通过，并将相关的信息存入 Current read-lock queue 中，而如果上面两个条件中任何一个没有满足，都会被迫进入等待队列Pending read-lock queue 中等待资源的释放。

写锁定

当客户端请求写锁定的时候，MySQL 首先检查在 Current write-lock queue 是否已经有锁定相同资源的信息存在。

如果 Current write-lock queue 没有，则再检查 Pending write-lock queue，如果在 Pending write-lock queue 中找到了，自己也需要进入等待队列并暂停自身线程等待锁定资源。反之，如果 Pending write-lock queue 为空，则再检测 Current read-lock queue，如果有锁定存在，则同样需要进入 Pending write-lock queue 等待。当然，也可能遇到以下这两种特殊情况：

1. 请求锁定的类型为 WRITE_DELAYED;
2. 请求锁定的类型为 WRITE_CONCURRENT_INSERT 或者是 TL_WRITE_ALLOW_WRITE，同时 Current read lock 是 READ_NO_INSERT 的锁定类型。

当遇到这两种特殊情况的时候，写锁定会立即获得而进入 Current write-lock queue 中

如果刚开始第一次检测就 Current write-lock queue 中已经存在了锁定相同资源的写锁定存在，那么就只能进入等待队列等待相应资源锁定的释放了。

读请求和写等待队列中的写锁请求的优先级规则主要为以下规则决定：

1. 除了 READ_HIGH_PRIORITY 的读锁定之外，Pending write-lock queue 中的 WRITE 写锁定能够阻塞所有其他的读锁定；
2. READ_HIGH_PRIORITY 读锁定的请求能够阻塞所有 Pending write-lock queue 中的写锁定；
3. 除了 WRITE 写锁定之外，Pending write-lock queue 中的其他任何写锁定都比读锁定的优先级低。

写锁定出现在 Current write-lock queue 之后，会阻塞除了以下情况下的所有其他锁定的请求：

1. 在某些存储引擎的允许下，可以允许一个 WRITE_CONCURRENT_INSERT 写锁定请求
2. 写锁定为 WRITE_ALLOW_WRITE 的时候，允许除了 WRITE_ONLY 之外的所有读和写锁定请求
3. 写锁定为 WRITE_ALLOW_READ 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求
4. 写锁定为 WRITE_DELAYED 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求
5. 写锁定为 WRITE_CONCURRENT_INSERT 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求

随着 MySQL 存储引擎的不断发展，目前 MySQL 自身提供的锁定机制已经没有办法满足需求了，很多存储引擎都在 MySQL 所提供的锁定机制之上做了存储引擎自己的扩展和改造。

MyISAM 存储引擎基本上可以说是对 MySQL 所提供的锁定机制所实现的表级锁定依赖最大的一种存储引擎了，虽然 MyISAM 存储引擎自己并没有在自身增加其他的锁定机制，但是为了更好的支持相关特性，

MySQL 在原有锁定机制的基础上为了支持其 Concurrent Insert 的特性而进行了相应的实现改造。

而其他几种支持事务的存储存储引擎，如 Innodb，NDB Cluster 以及 Berkeley DB 存储引擎则是让 MySQL 将锁定的处理直接交给存储引擎自己来处理，在 MySQL 中仅持有 WRITE_ALLOW_WRITE 类型的锁定。

由于 MyISAM 存储引擎使用的锁定机制完全是由 MySQL 提供的表级锁定实现，所以下面我们将以 MyISAM 存储引擎作为示例存储引擎，来实例演示表级锁定的一些基本特性。由于，为了让示例更加直观，我将使用显示给表加锁来演示：

<table>
<tr><th>时刻</th><th>Session a</th><th>Session b</th></tr>
<tr><td></td><td>READ</td><td></td></tr>
<tr><td>1</td><td>sky@localhost : example 11:21:08> lock table test_table_lock read;
Query OK, 0 rows affected (0.00 sec)
显示给 test_table_lock 加读锁定</td><td></td></tr>
<tr><td>2</td><td>sky@localhost : example 11:21:10> select * from test_table_lock limit 1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
+------+------+
1 row in set (0.01 sec)
自己的读操作未被阻塞</td><td>sky@localhost : example 11:21:13> select * from test_table_lock limit 1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
+------+------+
1 row in set (0.01 sec)
其他线程的读也未被阻塞</td></tr>
<tr><td>3</td><td>sky@localhost : example 11:21:15> update test_table_lock set b = a limit 1;
ERROR 1099 (HY000): Table 'test_table_lock' was locked with a READ lock and can't be updated</td><td>sky@localhost : example 11:21:20> update test_table_lock set b = a limit 1;
写一下试试看？被阻塞了</td></tr>
<tr><td>4</td><td>sky@localhost : example 11:21:09> unlock tables;
Query OK, 0 rows affected (0.00 sec)
解除读锁</td><td></td></tr>
<tr><td>5</td><td></td><td>sky@localhost : example 11:21:20> update test_table_lock set b = a limit 1;
Query OK, 0 rows affected (1 min 15.52 sec)
Rows matched: 1 Changed: 0 Warnings: 0
在 session a 释放锁定资源之后，session b 获得了资源，更新成功</td></tr>
<tr><td></td><td>sky@localhost : example 11:48:19> l</td><td>sky@localhost : example 11:48:20> ins</td></tr>
</table>

<table>
<tr><td></td><td>ock table test_table_lock read local;
Query OK, 0 rows affected (0.00 sec)
获取读锁定的时候增加 local 选项</td><td>ert into test_table_lock values(1,’s’,’c’);
Query OK, 1 row affected (0.00 sec)
其他 session 的 insert 未被阻塞</td></tr>
<tr><td></td><td></td><td>sky@localhost : example 11:48:23> update test_table_lock set a = 1 limit 1;
其他 session 的更新操作被阻塞</td></tr>
<tr><td></td><td>WRITE</td><td></td></tr>
<tr><td>6</td><td>这次加写锁试试看：
sky@localhost : example 11:27:01> lock table test_table_lock write;
Query OK, 0 rows affected (0.00 sec)</td><td></td></tr>
<tr><td>7</td><td>sky@localhost : example 11:27:10> select * from test_table_lock limit 1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
+------+------+
1 row in set (0.01 sec)
自己 session 可以继续读</td><td>sky@localhost : example 11:27:16> select * from test_table_lock limit 1;
其他 session 被阻塞</td></tr>
<tr><td>8</td><td>sky@localhost : example 11:27:02> unlock tables;
Query OK, 0 rows affected (0.00 sec)
释放锁定资源</td><td></td></tr>
<tr><td>9</td><td></td><td>sky@localhost : example 11:27:16> select * from test_table_lock limit 1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
+------+------+
1 row in set (36.36 sec)
其他 session 获取的资源</td></tr>
<tr><td></td><td>WRITE_ALLOW_READ</td><td></td></tr>
<tr><td>10</td><td>sky@localhost : example 11:42:24> alter table test_table_lock add(c varchar(16));
Query OK, 5242880 rows affected (7.06 sec)
Records: 5242880 Duplicates: 0 Warnings: 0

通过执行 DDL（ALTER TABLE），获取 W</td><td>sky@localhost : example 11:42:25> select * from test_table_lock limit 1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
+------+------+
1 row in set (0.01 sec)
其他 session 的读未被阻塞</td></tr>
</table>

	RITE_ALLOW_READ 类型的写锁定	

行级锁定

行级锁定不是 MySQL 自己实现的锁定方式，而是由其他存储引擎自己所实现的，如广为大家所知的 Innodb 存储引擎，以及 MySQL 的分布式存储引擎 NDB Cluster 等都是实现了行级锁定。

Innodb 锁定模式及实现机制

考虑到行级锁定君由各个存储引擎自行实现，而且具体实现也各有差别，而 Innodb 是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下 Innodb 的锁定特性。

总的来说，Innodb 的锁定机制和 Oracle 数据库有不少相似之处。Innodb 的行级锁定同样分为两种类型，共享锁和排他锁，而在锁定机制的实现过程中为了让行级锁定和表级锁定共存，Innodb 也同样使用了意向锁（表级锁定）的概念，也就有了意向共享锁和意向排他锁这两种。

当一个事务需要给自己需要的某个资源加锁的时候，如果遇到一个共享锁正锁定着自己需要的资源的时候，自己可以再加一个共享锁，不过不能加排他锁。但是，如果遇到自己需要锁定的资源已经被一个排他锁占有之后，则只能等待该锁定释放资源之后自己才能获取锁定资源并添加自己的锁定。 而意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被排他锁占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个共享锁，那么就在表上面添加一个意向共享锁。而如果自己需要的是某行（或者某些行）上面添加一个排他锁的话，则先在表上面添加一个意向排他锁。意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。所以，可以说 Innodb 的锁定模式实际上可以分为四种：共享锁（S），排他锁（X），意向共享锁（IS）和意向排他锁（IX），我们可以通过以下表格来总结上面这四种所的共存逻辑关系：

	共享锁（S）	排他锁（X）	意向共享锁（IS）	意向排他锁（IX）
共享锁（S）	兼容	冲突	兼容	冲突
排他锁（X）	冲突	冲突	冲突	冲突
意向共享锁（IS）	兼容	冲突	兼容	兼容
意向排他锁（IX）	冲突	冲突	兼容	兼容

虽然 Innodb 的锁定机制和 Oracle 有不少相近的地方，但是两者的实现确是截然不同的。总的来说就是 Oracle 锁定数据是通过需要锁定的某行记录所在的物理 block 上的事务槽上表级锁定信息，而 Innodb 的锁定则是通过在指向数据记录的第一个索引键之前和最后一个索引键之后的空域空间上标记锁定信息而实现的。Innodb 的这种锁定实现方式被称为“NEXT-KEY locking”（间隙锁），因为 Query 执行过程中通过过范围查找的华，他会锁定整个范围内所有的索引键值，即使这个键值并不存在。

间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害。而 Innodb 给出的解释是为了组织幻读的出现，所以他们选择的间隙锁来实现锁定。

除了间隙锁给 Innodb 带来性能的负面影响之外，通过索引实现锁定的方式还存在其他几个较大的性

能隐患：

- 当 Query 无法利用索引的时候，Innodb 会放弃使用行级别锁定而改用表级别的锁定，造成并发性能的降低；
- 当 Quuery 使用的索引并不包含所有过滤条件的时候，数据检索使用到的索引键所只想的数据可能有部分并不属于该 Query 的结果集的行列，但是也会被锁定，因为间隙锁锁定的是一个范围，而不是具体的索引键；
- 当 Query 在使用索引定位数据的时候，如果使用的索引键一样但访问的数据行不同的时候（索引只是过滤条件的一部分），一样会被锁定

Innodb 各事务隔离级别下锁定及死锁

Innodb 实现的在 ISO／ANSI SQL92 规范中所定义的 Read UnCommited，Read Commited，Repeatable Read 和 Serializable 这四种事务隔离级别。同时，为了保证数据在事务中的一致性，实现了多版本数据访问。

之前在第一节中我们已经介绍过，行级锁定肯定会带来死锁问题，Innodb 也不可能例外。至于死锁的产生过程我们就不在这里详细描述了，在后面的锁定示例中会通过一个实际的例子为大家爱展示死锁的产生过程。这里我们主要介绍一下，在 Innodb 中当系检测到死锁产生之后是如何来处理的。

在 Innodb 的事务管理和锁定机制中，有专门检测死锁的机制，会在系统中产生死锁之后的很短时间内就检测到该死锁的存在。当 Innodb 检测到系统中产生了死锁之后，Innodb 会通过相应的判断来选这产生死锁的两个事务中较小的事务来回滚，而让另外一个较大的事务成功完成。那 Innodb 是以什么来为标准判定事务的大小的呢？MySQL 官方手册中也提到了这个问题，实际上在 Innodb 发现死锁之后，会计算出两个事务各自插入、更新或者删除的数据量来判定两个事务的大小。也就是说哪个事务所改变的记录条数越多，在死锁中就越不会被回滚掉。但是有一点需要注意的就是，当产生死锁的场景中涉及到不止 Innodb 存储引擎的时候，Innodb 是没办法检测到该死锁的，这时候就只能通过锁定超时限制来解决该死锁了。另外，死锁的产生过程的示例将在本节最后的 Innodb 锁定示例中演示。

Innodb 锁定机制示例

```
mysql> create table test_innodb_lock (a int(11),b varchar(16)) engine=innodb;
Query OK, 0 rows affected (0.02 sec)

mysql> create index test_innodb_a_ind on test_innodb_lock(a);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> create index test_innodb_lock_b_ind on test_innodb_lock(b);
Query OK, 11 rows affected (0.01 sec)
Records: 11  Duplicates: 0  Warnings: 0
```

时刻	Session a	Session b
	行锁定基本演示	
1	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)

	mysql> update test_innodb_lock set b = 'b1' where a = 1; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 更新，但是不提交	
2		mysql> update test_innodb_lock set b = 'b1' where a = 1; 被阻塞，等待
3	mysql> commit; Query OK, 0 rows affected (0.05 sec) 提交	
4		mysql> update test_innodb_lock set b = 'b1' where a = 1; Query OK, 0 rows affected (36.14 sec) Rows matched: 1 Changed: 0 Warnings: 0 解除阻塞，更新正常进行
	无索引升级为表锁演示	
5	mysql> update test_innodb_lock set b = '2' where b = 2000; Query OK, 1 row affected (0.02 sec) Rows matched: 1 Changed: 1 Warnings: 0	
6		mysql> update test_innodb_lock set b = '3' where b = 3000; 被阻塞，等待
7	mysql> commit; Query OK, 0 rows affected (0.10 sec)	
8		mysql> update test_innodb_lock set b = '3' where b = 3000; Query OK, 1 row affected (1 min 3.41 sec) Rows matched: 1 Changed: 1 Warnings: 0 阻塞解除，完成更新
	间隙锁带来的插入问题演示	
9	mysql> select * from test_innodb_lock; +------+------+ | a	b		

	+------+------+ | 1 | b2 | | 3 | 3 | | 4 | 4000 | | 5 | 5000 | | 6 | 6000 | | 7 | 7000 | | 8 | 8000 | | 9 | 9000 | | 1 | b1 | +------+------+ 9 rows in set (0.00 sec) mysql> update test_innodb_lock set b = a * 100 where a < 4 and a > 1; Query OK, 1 row affected (0.02 sec) Rows matched: 1 Changed: 1 Warnings: 0	
10		mysql> insert into test_innodb_lock values(2,'200'); 被阻塞，等待
11	mysql> commit; Query OK, 0 rows affected (0.02 sec)	
12		mysql> insert into test_innodb_lock values(2,'200'); Query OK, 1 row affected (38.68 sec) 阻塞解除，完成插入
	使用共同索引不同数据的阻塞示例	
13	mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b2'; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0	
14		mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b1'; 被阻塞
15	mysql> commit; Query OK, 0 rows affected (0.02 sec)	

16		mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b1'; Query OK, 1 row affected (42.89 sec) Rows matched: 1 Changed: 1 Warnings: 0 session 提交事务，阻塞去除，更新完成
	死锁示例	
17	mysql> update t1 set id = 110 where id = 11; Query OK, 0 rows affected (0.00 sec) Rows matched: 0 Changed: 0 Warnings: 0	
18		mysql> update t2 set id = 210 where id = 21; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0
19	mysql> update t2 set id = 2100 where id = 21; 等待 session b 释放资源，被阻塞	
20		mysql> update t1 set id = 1100 where id = 11; Query OK, 0 rows affected (0.39 sec) Rows matched: 0 Changed: 0 Warnings: 0 等待 session a 释放资源，被阻塞
	两个 session 互相等等待对方的资源释放之后才能释放自己的资源，造成了死锁	

7．3 合理利用锁机制优化 MySQL

MyISAM 表锁优化建议

对于 MyISAM 存储引擎，虽然使用表级锁定在锁定实现的过程中比实现行级锁定或者页级锁所带来的附加成本都要小，锁定本身所消耗的资源也是最少。但是由于锁定的颗粒度比较到，所以造成锁定资源的争用情况也会比其他的锁定级别都要多，从而在较大程度上会降低并发处理能力。

所以，在优化 MyISAM 存储引擎锁定问题的时候，最关键的就是如何让其提高并发度。由于锁定级别是不可能改变的了，所以我们首先需要尽可能让锁定的时间变短，然后就是让可能并发进行的操作尽可能的并发。

1、 缩短锁定时间

缩短锁定时间，短短几个字，说起来确实听容易的，但实际做起来恐怕就并不那么简单了。如何让锁定时间尽可能的短呢？唯一的办法就是让我们的 Query 执行时间尽可能的短。

a) 尽两减少大的复杂 Query，将复杂 Query 分拆成几个小的 Query 分布进行；
b) 尽可能的建立足够高效的索引，让数据检索更迅速；
c) 尽量让 MyISAM 存储引擎的表只存放必要的信息，控制字段类型；
d) 利用合适的机会优化 MyISAM 表数据文件；

2、 分离能并行的操作

说到MyISAM的表锁，而且是读写互相阻塞的表锁，可能有些人会认为在MyISAM存储引擎的表上就只能是完全的串行化，没办法再并行了。大家不要忘记了，MyISAM 的存储引擎还有一个非常有用的特性，那就是 Concurrent Insert（并发插入）的特性。

MyISAM 存储引擎有一个控制是否打开 Concurrent Insert 功能的参数选项：concurrent_insert，可以设置为 0，1 或者 2。三个值的具体说明如下：

a) concurrent_insert=2，无论 MyISAM 存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都允许在数据文件尾部进行 Concurrent Insert;
b) concurrent_insert=1，当 MyISAM 存储引擎表数据文件中间不存在空闲空间的时候，可以从文件尾部进行 Concurrent Insert;
c) concurrent_insert=0，无论 MyISAM 存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都不允许 Concurrent Insert。

3、合理利用读写优先级

在本章各种锁定分析一节中我们了解到了 MySQL 的表级锁定对于读和写是有不同优先级设定的，默认情况下是写优先级要大于读优先级。所以，如果我们可以根据各自系统环境的差异决定读与写的优先级。如果我们的系统是一个以读为主，而且要优先保证查询性能的话，我们可以通过设置系统参数选项 low_priority_updates=1，将写的优先级设置为比读的优先级低，即可让告诉 MySQL 尽量先处理读请求。当然，如果我们的系统需要有限保证数据写入的性能的话，则可以不用设置 low_priority_updates 参数了。

这里我们完全可以利用这个特性，将 concurrent_insert 参数设置为 1，甚至如果数据被删除的可能性很小的时候，如果对暂时性的浪费少量空间并不是特别的在乎的话，将 concurrent_insert 参数设置为 2 都可以尝试。当然，数据文件中间留有空域空间，在浪费空间的时候，还会造成在查询的时候需要读取更多的数据，所以如果删除量不是很小的话，还是建议将 concurrent_insert 设置为 1 更为合适。

Innodb 行锁优化建议

Innodb 存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一些，但是在整体并发处理能力方面要远远优于 MyISAM 的表级锁定的。当系统并发量较高的时候，Innodb 的整体性能和 MyISAM 相比就会有比较明显的优势了。但是，Innodb 的行级锁定同样也有其脆弱的一面，当我们使用不当的时候，可能会让 Innodb 的整体性能表现不仅不能比 MyISAM 高，甚至可能会更差。

要想合理利用 Innodb 的行级锁定，做到扬长避短，我们必须做好以下工作：

a) 尽可能让所有的数据检索都通过索引来完成，从而避免 Innodb 因为无法通过索引键加锁而升级为表级锁定；

b) 合理设计索引，让 Innodb 在索引键上面加锁的时候尽可能准确，尽可能的缩小锁定范围，避免造成不必要的锁定而影响其他 Query 的执行；
c) 尽可能减少基于范围的数据检索过滤条件，避免因为间隙锁带来的负面影响而锁定了不该锁定的记录；
d) 尽量控制事务的大小，减少锁定的资源量和锁定时间长度；
e) 在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少 MySQL 因为实现事务隔离级别所带来的附加成本；

由于 Innodb 的行级锁定和事务性，所以肯定会产生死锁，下面是一些比较常用的减少死锁产生概率的的小建议，读者朋友可以根据各自的业务特点针对性的尝试：
a) 类似业务模块中，尽可能按照相同的访问顺序来访问，防止产生死锁；
b) 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；
c) 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

系统锁定争用情况查询

对于两种锁定级别，MySQL 内部有两组专门的状态变量记录系统内部锁资源争用情况，我们先看看 MySQL 实现的表级锁定的争用状态变量：

```
mysql> show status like 'table%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Table_locks_immediate | 100   |
| Table_locks_waited    | 0     |
+-----------------------+-------+
```

这里有两个状态变量记录 MySQL 内部表级锁定的情况，两个变量说明如下：

- Table_locks_immediate：产生表级锁定的次数；
- Table_locks_waited：出现表级锁定争用而发生等待的次数；

两个状态值都是从系统启动后开始记录，没出现一次对应的事件则数量加 1。如果这里的 Table_locks_waited 状态值比较高，那么说明系统中表级锁定争用现象比较严重，就需要进一步分析为什么会有较多的锁定资源争用了。

对于 Innodb 所使用的行级锁定，系统中是通过另外一组更为详细的状态变量来记录的，如下：

```
mysql> show status like 'innodb_row_lock%';
+-------------------------------+--------+
| Variable_name                 | Value  |
+-------------------------------+--------+
| Innodb_row_lock_current_waits | 0      |
| Innodb_row_lock_time          | 490578 |
| Innodb_row_lock_time_avg      | 37736  |
| Innodb_row_lock_time_max      | 121411 |
| Innodb_row_lock_waits         | 13     |
+-------------------------------+--------+
```

Innodb 的行级锁定状态变量不仅记录了锁定等待次数，还记录了锁定总时长，每次平均时长，以及最大时长，此外还有一个非累积状态量显示了当前正在等待锁定的等待数量。对各个状态量的说明如下：

- Innodb_row_lock_current_waits：当前正在等待锁定的数量；
- Innodb_row_lock_time：从系统启动到现在锁定总时间长度；
- Innodb_row_lock_time_avg：每次等待所花平均时间；
- Innodb_row_lock_time_max：从系统启动到现在等待最常的一次所花的时间；
- Innodb_row_lock_waits：系统启动后到现在总共等待的次数；

对于这 5 个状态变量，比较重要的主要是 Innodb_row_lock_time_avg（等待平均时长），Innodb_row_lock_waits（等待总次数）以及 Innodb_row_lock_time（等待总时长）这三项。尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

此外，Innodb 出了提供这五个系统状态变量之外，还提供的其他更为丰富的即时状态信息供我们分析使用。可以通过如下方法查看：

1. 通过创建 Innodb Monitor 表来打开 Innodb 的 monitor 功能：

```
mysql> create table innodb_monitor(a int) engine=innodb;
Query OK, 0 rows affected (0.07 sec)
```

2. 然后通过使用“SHOW INNODB STATUS”查看细节信息（由于输出内容太多就不在此记录了）；

可能会有读者朋友问为什么要先创建一个叫 innodb_monitor 的表呢？因为创建该表实际上就是告诉 Innodb 我们开始要监控他的细节状态了，然后 Innodb 就会将比较详细的事务以及锁定信息记录进入 MySQL 的 error log 中，以便我们后面做进一步分析使用。

7．4 小结

本章以 MySQL Server 中的锁定简介开始，分析了当前 MySQL 中使用最为广泛的锁定方式表级锁定和行级锁定的基本实现机制，并通过 MyISAM 和 Innodb 这两大典型的存储引擎作为示例存储引擎所使用的表级锁定和行级锁定做了较为详细的分析和演示。然后，再通过分析两种锁定方式的特性，给出相应的优化建议和策略。最后了解了一下在 MySQL Server 中如何获得系统当前各种锁定的资源争用状况。希望本章内容能够对各位读者朋友在理解 MySQL 锁定机制方面有一定的帮助。

第 8 章 MySQL 数据库 Query 的优化

前言：

在之前“影响 MySQL 应用系统性能的相关因素”一章中我们就已经分析过了 Query 语句对数据库性能的影响非常大，所以本章将专门针对 MySQL 的 Query 语句的优化进行相应的分析。

8.1 理解 MySQL 的 Query Optimizer

8.1.1 MySQL Query Optimizer 是什么？

在“MySQL 架构组成”一章中的 “MySQL 逻辑组成”一节中我们已经了解到，在 MySQL 中有一个专门负责优化 SELECT 语句的优化器模块，这就是我们本节将要重点分析的 MySQL Optimizer，其主要的功能就是通过计算分析系统中收集的各种统计信息，为客户端请求的 Query 给出他认为最优的执行计划，也就是他认为最优的数据检索方式。

当 MySQL Optimizer 接收到从 Query Parser （解析器）送过来的 Query 之后，会根据 MySQL Query 语句的相应语法对该 Query 进行分解分析的同时，还会做很多其他的计算转化工作。如常量转化，无效内容删除，常量计算等等。所有这些工作都只为了 Optimizer 工作的唯一目的，分析出最优的数据检索方式，也就是我们常说的执行计划。

8.1.2 MySQL Query Optimizer 基本工作原理

在分析 MySQL Optimizer 的工作原理之前，先了解一下 MySQL 的 Query Tree。MySQL 的 Query Tree 是通过优化实现 DBXP 的经典数据结构和 Tree 构造器而生成的一个指导完成一个 Query 语句的请求所需要处理的工作步骤，我们可以简单的认为就是一个的数据处理流程规划，只不过是以一个 Tree 的数据结构存放而已。通过 Query Tree 我们可以很清楚的知道一个 Query 的完成需要经过哪些步骤的处理，每一步的数据来源在哪里，处理方式是怎样的。在整个 DBXP 的 Query Tree 生成过程中，MySQL 使用了 LEX 和 YACC 这两个功能非常强大的语法（词法）分析工具。MySQL Query Optimizer 的所有工作都是基于这个 Query Tree 所进行的。各位读者朋友如果对 MySQL Query Tree 实现生成的详细信息比较感兴趣，可以参考 Chales A. Bell 的《Expert MySQL》这本书，里面有比较详细的介绍。

MySQL Query Optimizer 并不是一个纯粹的 CBO（Cost Base Optimizer），而是在 CBO 的基础上增加了一个被称为 Heuristic Optimize（启发式优化）的功能。也就是说，MySQL Query Optimizer 在优化一个 Query 选择出他认为的最优执行计划的时候，并不一定完全按照系数据库的元信息和系统统计信息，而是在此基础上增加了某些特定的规则。其实我个人的理解就是在 CBO 的实现中增加了部分 RBO（Rule Base Optimizer）的功能，以确保在某些特别的场景下控制 Query 按照预定的方式生成执行计划。

当客户端向 MySQL 请求一条 Query ，到命令解析器模块完成请求分类区别出是 SELECT 并转发给 Query Optimizer 之后，Query Optimizer 首先会对整条 Query 进行，优化处理掉一些常量表达式的预算，直接换算成常量值。并对 Query 中的查询条件进行简化和转换，如去掉一些无用或者显而易见的条

件，结构调整等等。然后则是分析 Query 中的 Hint 信息（如果有），看显示 Hint 信息是否可以完全确定该 Query 的执行计划。如果没有 Hint 或者 Hint 信息还不足以完全确定执行计划，则会读取所涉及对象的统计信息，根据 Query 进行写相应的计算分析，然后再得出最后的执行计划。

Query Optimizer 是一个数据库软件非常核心的功能，虽然在这里说起来只是简单的几句话，但是在 MySQL 内部，Query Optimizer 实际上是经过了很多复杂的运算分析，才得出最后的执行计划。对于 MySQL Query Optimizer 更多的信息，各位读者可以通过 MySQL Internal 文档进行更为全面的了解。

8.2 Query 语句优化基本思路和原则

在分析如何优化 MySQL Query 之前，我们需要先了解一下 Query 语句优化的基本思路和原则。一般来说，Query 语句的优化思路和原则主要提现在以下几个方面：

1. 优化更需要优化的 Query；
2. 定位优化对象的性能瓶颈；
3. 明确的优化目标；
4. 从 Explain 入手；
5. 多使用 profile
6. 永远用小结果集驱动大的结果集；
7. 尽可能在索引中完成排序；
8. 只取出自己需要的 Columns；
9. 仅仅使用最有效的过滤条件；
10. 尽可能避免复杂的 Join 和子查询；

上面所列的几点信息，前面 4 点可以理解为 Query 优化的一个基本思路，后面部分则是我们优化中的基本原则。

下面我们先针对 Query 优化的基本思路做一些简单的分析，理解为什么我们的 Query 优化到底该如何进行。

优化更需要优化的 Query

为什么我们需要优化更需要优化的 Query？这个地球人都知道的“并不能成为问题的问题”我想就并不需要我过多解释吧，哈哈。

那什么样的 Query 是更需要优化呢？对于这个问题我们需要从对整个系统的影响来考虑。什么 Query 的优化能给系统整体带来更大的收益，就更需要优化。一般来说，高并发低消耗（相对）的 Query 对整个系统的影响远比低并发高消耗的 Query 大。我们可以通过以下一个非常简单的案例分析来充分说明问题。

假设有一个 Query 每小时执行 10000 次，每次需要 20 个 IO。另外一个 Query 每小时执行 10 次，每次需要 20000 个 IO。

我们先通过IO消耗方面来分析。可以看出，两个 Query 每小时所消耗的 IO总数目是一样的，都是 200000 IO/小时。假设我们优化第一个 Query，从20个IO降低到18个IO，也就是仅仅降低了 2 个IO，则我们节省了2 * 10000 = 20000 （IO/小时）。而如果希望通过优化第二个 Query达到相同的效果，我们必须要让每个 Query 减少 20000 / 10 = 2000 IO。我想大家都会相信让第一个 Query 节省2个IO远比第二个 Query 节省2000个IO来的容易。

其次，如果通过 CPU 方面消耗的比较，原理和上面的完全一样。只要让第一个 Query 稍微节省一小块资源，就可以让整个系统节省出一大块资源，尤其是在排序，分组这些对 CPU 消耗比较多的操作中尤其突出。

最后，我们从对整个系统的影响来分析。一个频繁执行的高并发 Query 的危险性比一个低并发的 Query 要大很多。当一个低并发的 Query 走错执行计划，所带来的影响主要只是该 Query 的请求者的体验会变差，对整体系统的影响并不会特别的突出，之少还属于可控范围。但是，如果我们一个高并发的 Query 走错了执行计划，那所带来的后可很可能就是灾难性的，很多时候可能连自救的机会都不给你就会让整个系统 Crash 掉。曾经我就遇到这样一个案例，系统中一个并发度较高的 Query 语句走错执行计划，系统顷刻间 Crash，甚至我都还没有反应过来是怎么回事。当重新启动数据库提供服务后，系统负载立刻直线飙升，甚至都来不及登录数据库查看当时有哪些 Active 的线程在执行哪些 Query。如果是遇到一个并发并不太高的 Query 走错执行计划，至少我们还可以控制整个系统不至于系统被直接压跨，甚至连问题根源都难以抓到。

定位优化对象的性能瓶颈

当我们拿到一条需要优化的 Query 之后，第一件事情是什么？是反问自己，这条 Query有什么问题？我为什么要优化他？只有明白了这些问题，我们才知道我们需要做什么，才能够找到问题的关键。而不能就只是觉得某个 Query 好像有点慢，需要优化一下，然后就开始一个一个优化方法去轮番尝试。这样很可能整个优化过程会消耗大量的人力和时间成本，甚至可能到最后还是得不到一个好的优化结果。这就像看病一样，医生必须要清楚的知道我们病的根源才能对症下药。如果只是知道我们什么地方不舒服，然后就开始通过各种药物尝试治疗，那这样所带来的后果可能就非常严重了。

所以，在拿到一条需要优化的 Query 之后，我们首先要判断出这个 Query 的瓶颈到底是IO还是CPU。到底是因为在数据访问消耗了太多的时间，还是在数据的运算（如分组排序等）方面花费了太多资源？

一般来说，在MySQL 5.0系列版本中，我们可以通过系统自带的 PROFILING功能很清楚的找出一个 Query 的瓶颈所在。当然，如果读者朋友为了使用 MySQL 的某些在 5.1 版本中才有的新特性（如 Partition，EVENT等）亦或者是比较喜欢尝试新事务而早早使用的 MySQL 5.1 的预发布版本，可能就没办法使用这个功能了，因为该功能在MySQL5.1 系列刚开始的版本中并不支持，不过让人非常兴奋的是该功能在最新出来的 MySQL 5.1 正式版（5.1.30）又已经提供了。而如果读者朋友正在使用的MySQL是4.x版本，那可能就只能通过自行分析 Query 的各个执行步骤，找到性能损失最大的地方。

明确的优化目标

当我们定为到了一条 Query 的性能瓶颈之后，就需要通过分析该 Query 所完成的功能和 Query 对系统的整体影响制订出一个明确的优化目标。没有一个明确的目标，优化过程将是一个漫无目的而且低

效的过程，也很难达收到一个理想的效果。尤其是对于一些实现应用中较为重要功能点的 Query 更是如此。

如何设定优化目标？这可能是很多人都非常头疼的问题，对于我自己也一样。要设定一个合理的优化目标，不能过于理想也不能放任自由，确实是一件非常头疼的事情。一般来说，我们首先需要清楚的了解数据库目前的整体状态，同时也要清楚的知道数据库中与该 Query 相关的数据库对象的各种信息，而且还要了解该 Query 在整个应用系统中所实现的功能。了解了数据库整体状态，我们就能知道数据库所能承受的最大压力，也就清楚了我们能够接受的最悲观情况。把握了该 Query 相关数据库对象的信息，我们就应该知道实现该 Query 的消耗最理想情况下需要消耗多少资源，最糟糕又需要消耗多少资源。最后，通过该 Query 所实现的功能点在整个应用系统中的重要地位，我们可以大概的分析出该 Query 可以占用的系统资源比例，而且我们也能够知道该 Query 的效率给客户带来的体验影响到底有多大。

当我们清楚了这些信息之后，我们基本可以得出该 Query 应该满足的一个性能范围是怎样的，这也就是我们的优化目标范围，然后就是通过寻找相应的优化手段来解决问题了。如果该 Query 实现的应用系统功能比较重要，我们就必须让目标更偏向于理想值一些，即使在其他某些方面作出一些让步与牺牲，比如调整 schema 设计，调整索引组成等，可能都是需要的。而如果该 Query 所实现的是一些并不是太关键的功能，那我们可以让目标更偏向悲观值一些，而尽量保证其他更重要的 Query 的性能。这种时候，即使需要调整商业需求，减少功能实现，也不得不应该作出让步。

从 Explain 入手

现在，优化目标也已经明确了，自然是奥开始动手的时候了。我们的优化到底该从何处入手呢？答案只有一个，从 Explain 开始入手。为什么？因为只有 Explain 才能告诉你，这个 Query 在数据库中是以一个什么样的执行计划来实现的。

但是，有一点我们必须清楚，Explain 只是用来获取一个 Query 在当前状态的数据库中的执行计划，在优化动手之前，我们比需要根据优化目标在自己头脑中有一个清晰的目标执行计划。只有这样，优化的目标才有意义。一个优秀的 SQL 调优人员（或者成为 SQL Performance Tuner），在优化任何一个 SQL 语句之前，都应该在自己头脑中已经先有一个预定的执行计划，然后通过不断的调整尝试，再借助 Explain 来验证调整的结果是否满足自己预定的执行计划。对于不符合预期的执行计划需要不断分析 Query 的写法和数据库对象的信息，继续调整尝试，直至得到预期的结果。

当然，人无完人，并不一定每次自己预设的执行计划都肯定是最优的，在不断调整测试的过程中，如果发现 MySQL Optimizer 所选择的执行计划的实际执行效果确实比自己预设的要好，我们当然还是应该选择使用 MySQL optimizer 所生成的执行计划。

上面的这个优化思路，只是给大家指了一个优化的基本方向，实际操作还需要读者朋友不断的结合具体应用场景不断的测试实践来体会。当然也并不一定所有的情况都非要严格遵循这样一个思路，规则是死的，人是活的，只有更合理的方法，没有最合理的规则。

在了解了上面这些优化的基本思路之后，我们再来看看优化的几个基本原则。

永远用小结果集驱动大的结果集

很多人喜欢在优化 SQL 的时候说用小表驱动大表，个人认为这样的说法不太严谨。为什么？因为大表经过 WHERE 条件过滤之后所返回的结果集并不一定就比小表所返回的结果集大，可能反而更小。在这种情况下如果仍然采用小表驱动大表，就会得到相反的性能效果。

其实这样的结果也非常容易理解，在 MySQL 中的 Join，只有 Nested Loop 一种 Join 方式，也就是 MySQL 的 Join 都是通过嵌套循环来实现的。驱动结果集越大，所需要循环的此时就越多，那么被驱动表的访问次数自然也就越多，而每次访问被驱动表，即使需要的逻辑 IO 很少，循环次数多了，总量自然也不可能很小，而且每次循环都不能避免的需要消耗 CPU ，所以 CPU 运算量也会跟着增加。所以，如果我们仅仅以表的大小来作为驱动表的判断依据，假若小表过滤后所剩下的结果集比大表多很多，结果就是需要的嵌套循环中带来更多的循环次数，反之，所需要的循环次数就会更少，总体 IO 量和 CPU 运算量也会少。而且，就算是非 Nested Loop 的 Join 算法，如 Oracle 中的 Hash Join，同样是小结果集驱动大的结果集是最优的选择。

所以，在优化 Join Query 的时候，最基本的原则就是“小结果集驱动大结果集”，通过这个原则来减少嵌套循环中的循环次数，达到减少 IO 总量以及 CPU 运算的次数。

尽可能在索引中完成排序

只取出自己需要的 Columns

任何时候在 Query 中都只取出自己需要的 Columns，尤其是在需要排序的 Query 中。为什么？

对于任何 Query，返回的数据都是需要通过网络数据包传回给客户端，如果取出的 Column 越多，需要传输的数据量自然会越大，不论是从网络带宽方面考虑还是从网络传输的缓冲区来看，都是一个浪费。

如果是需要排序的 Query 来说，影响就更大了。在 MySQL 中存在两种排序算法，一种是在 MySQL4.1 之前的老算法，实现方式是先将需要排序的字段和可以直接定位到相关行数据的指针信息取出，然后在我们所设定的排序区（通过参数 sort_buffer_size 设定）中进行排序，完成排序之后再次通过行指针信息取出所需要的 Columns，也就是说这种算法需要访问两次数据。第二种排序算法是从 MySQL4.1 版本开始使用的改进算法，一次性将所需要的 Columns 全部取出，在排序区中进行排序后直接将数据返回给请求客户端。改行算法只需要访问一次数据，减少了大量的随机 IO，极大的提高了带有排序的 Query 语句的效率。但是，这种改进后的排序算法需要一次性取出并缓存的数据比第一种算法要多很多，如果我们将并不需要的 Columns 也取出来，就会极大的浪费排序过程所需要的内存。在 MySQL4.1 之后的版本中，我们可以通过设置 max_length_for_sort_data 参数大小来控制 MySQL 选择第一种排序算法还是第二种排序算法。当所取出的 Columns 的单条记录总大小 max_length_for_sort_data 设置的大小的时候，MySQL 就会选择使用第一种排序算法，反之，则会选择第二种优化后的算法。为了尽可能提高排序性能，我们自然是更希望使用第二种排序算法，所以在 Query 中仅仅取出我们所需要的 Columns 是非常有必要的。

仅仅使用最有效的过滤条件

很多人在优化 Query 语句的时候很容易进入一个误区，那就是觉得 WHERE 子句中的过滤条件越多越好，实际上这并不是一个非常正确的选择。其实我们分析 Query 语句的性能优劣最关键的就是要让他

选择一条最佳的数据访问路径，如何做到通过访问最少的数据量完成自己的任务。

为什么说过滤条件多不一定是好事呢？请看下面示例：

需求：　查找某个用户在所有 group 中所发的讨论 message 基本信息。

场景：　1、知道用户 ID 和用户 nick_name

　　　　2、信息所在表为 group_message

　　　　3、group_message 中存在用户 ID(user_id)和 nick_name(author)两个索引

方案一：将用户 ID 和用户 nick_name 两者都作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:29:37> EXPLAIN SELECT * FROM group_message
    ->  WHERE user_id = 1 AND author='1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind,group_message_uid_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

方案二：仅仅将用户 ID 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:30:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_uid_ind
          key: group_message_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

方案二：仅将用户 nick_name 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:38:45> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

初略一看三个执行计划好像都挺好的啊，每一个 Query 的执行类型都利用到了索引，而且都是“ref”类型。可是仔细一分析，就会发现，group_message_uid_ind 索引的索引键长度为 4（key_len: 4），由于 user_id 字段类型为 int，所以我们可以判定出 Query Optimizer 给出的这个索引键长度是完全准确的。而 group_message_author_ind 索引的索引键长度为 98（key_len: 98），因为 author 字段定义为 varchar(32) ，而所使用的字符集是 utf8，32 * 3 + 2 = 98。而且，由于 user_id 与 author（来源于 nick_name）全部都是一一对应的，所以同一个 user_id 有哪些记录，那么所对应的 author 也会有完全相同的记录。所以，同样的数据在 group_message_author_ind 索引中所占用的存储空间要远远大于 group_message_uid_ind 索引所占用的空间。占用空间更大，代表我们访问该索引所需要读取的数据量就会更多。所以，选择 group_message_uid_ind 的执行计划才是最有的执行计划。也就是说，上面的方案二才是最有方案，而使用了更多的 WHERE 条件的方案一反而没有仅仅使用 user_id 一个过滤条件的方案一优。

可能有些人会说，那如果将 user_id 和 author 两者建立联合索引呢？告诉你，效果可能比没有这个索引的时候更差，因为这个联合索引的索引键更长，索引占用的空间将会更大。

这个示例并不一定能代表所有场景，仅仅是希望让大家明白，并不是任何时候都是使用的过滤条件越多性能会越好。在实际应用场景中，肯定会存在更多更复杂的情形，怎样使我们的 Query 有一个更优化的执行计划，更高效的性能，还需要靠大家仔细分析各种执行计划的具体差别，才能选择出更优化的 Query。

尽可能避免复杂的 Join 和子查询

我们都知道，MySQL 在并发这一块做的并不是太好，当并发量太高的时候，系统整体性能可能会急剧下降，尤其是遇到一些较为复杂的 Query 的时候更是如此。这主要与 MySQL 内部资源的争用锁定控制有关，如读写相斥等等。对于 Innodb 存储引擎由于实现了行级锁定可能还要稍微好一些，如果使用的 MyISAM 存储引擎，并发一旦较高的时候，性能下降非常明显。所以，我们的 Query 语句所涉及到的表越多，所需要锁定的资源就越多。也就是说，越复杂的 Join 语句，所需要锁定的资源也就越多，所阻塞的其他线程也就越多。相反，如果我们将比较复杂的 Query 语句分拆成多个较为简单的 Query 语

句分步执行，每次锁定的资源也就会少很多，所阻塞的其他线程也要少一些。

可能很多读者会有疑问，将复杂 Join 语句分拆成多个简单的 Query 语句之后，那不是我们的网络交互就会更多了吗？网络延时方面的总体消耗也就更大了啊，完成整个查询的时间不是反而更长了吗？是的，这种情况是可能存在，但也并不是肯定就会如此。我们可以再分析一下，一个复杂的 Join Query 语句在执行的时候，所需要锁定的资源比较多，可能被别人阻塞的概率也就更大，如果是一个简单的 Query，由于需要锁定的资源较少，被阻塞的概率也会小很多。所以 较为复杂的 Join Query 也有可能在执行之前被阻塞而浪费更多的时间。而且，我们的数据库所服务的并不是单单这一个 Query 请求，还有很多很多其他的请求，在高并发的系统中，牺牲单个 Query 的短暂响应时间而提高整体处理能力也是非常值得的。优化本身就是一门平衡与取舍的艺术，只有懂得取舍，平衡整体，才能让系统更优。

对于子查询，可能不需要我多说很多人就明白为什么会不被推荐使用。在 MySQL 中，子查询的实现目前还比较差，很难得到一个很好的执行计划，很多时候明明有索引可以利用，可 Query Optimizer 就是不用。从 MySQL 官方给出的信息说，这一问题将在 MySQL6.0 中得到较好的解决，将会引入 SemiJoin 的执行计划，可 MySQL6.0 离我们投入生产环境使用恐怕还有很遥远的一段时间。所以，在 Query 优化的过程中，能不用子查询的时候就尽量不要使用子查询。

上面这些仅仅只是一些常用的优化原则，并不是说在 Query 优化中就只需要做到这些原则就可以，更不是说 Query 优化只能通过这些原则来优化。在实际优化过程中，我们还可能会遇到很多带有较为复杂商业逻辑的场景，具体的优化方法就只能根据不同的应用场景来具体分析，逐步调整。其实，最有效的优化，就是不要用，也就是不要实现这个商业需求。

8.3 充分利用 Explain 和 Profiling

8.3.1 Explain 的使用

说到 Explain，肯定很多读者之前都都已经用过了，MySQL Query Optimizer 通过我让们执行 EXPLAIN 命令来告诉我们他将使用一个什么样的执行计划来优化我们的 Query。所以，可以说 Explain 是在优化 Query 时最直接有效的验证我们想法的工具。在本章前面部分我就说过，一个好的 SQL Performance Tuner 在动手优化一个 Query 之前，头脑中就应该已经有一个好的执行计划，后面的优化工作只是为实现该执行计划而作出各种调整。

在我们对某个 Query 优化过程中，需要不断的使用 Explain 来验证我们的各种调整是否有效。就像本书之前的很多示例都会通过 Explain 来验证和展示结果一样，所有的 Query 优化都应该充分利用他。

我们先看一下在 MySQL Explain 功能中给我们展示的各种信息的解释：

- ◆ ID：Query Optimizer 所选定的执行计划中查询的序列号；
- ◆ Select_type：所使用的查询类型，主要有以下这几种查询类型
  - ◇ DEPENDENT SUBQUERY：子查询中内层的第一个 SELECT，依赖于外部查询的结果集；
  - ◇ DEPENDENT UNION：子查询中的 UNION，且为 UNION 中从第二个 SELECT 开始的后面所有

SELECT，同样依赖于外部查询的结果集；
  - ◇ PRIMARY：子查询中的最外层查询，注意并不是主键查询；
  - ◇ SIMPLE：除子查询或者 UNION 之外的其他查询；
  - ◇ SUBQUERY：子查询内层查询的第一个 SELECT，结果不依赖于外部查询结果集；
  - ◇ UNCACHEABLE SUBQUERY：结果集无法缓存的子查询；
  - ◇ UNION：UNION 语句中第二个 SELECT 开始的后面所有 SELECT，第一个 SELECT 为 PRIMARY
  - ◇ UNION RESULT：UNION 中的合并结果；
- ◆ Table：显示这一步所访问的数据库中的表的名称；
- ◆ Type：告诉我们对表所使用的访问方式，主要包含如下集中类型；
  - ◇ all：全表扫描
  - ◇ const：读常量，且最多只会有一条记录匹配，由于是常量，所以实际上只需要读一次；
  - ◇ eq_ref：最多只会有一条匹配结果，一般是通过主键或者唯一键索引来访问；
  - ◇ fulltext：
  - ◇ index：全索引扫描；
  - ◇ index_merge：查询中同时使用两个（或更多）索引，然后对索引结果进行 merge 之后再读取表数据；
  - ◇ index_subquery：子查询中的返回结果字段组合是一个索引（或索引组合），但不是一个主键或者唯一索引；
  - ◇ rang：索引范围扫描；
  - ◇ ref：Join 语句中被驱动表索引引用查询；
  - ◇ ref_or_null：与 ref 的唯一区别就是在使用索引引用查询之外再增加一个空值的查询；
  - ◇ system：系统表，表中只有一行数据；
  - ◇ unique_subquery：子查询中的返回结果字段组合是主键或者唯一约束；
  - ◇
- ◆ Possible_keys：该查询可以利用的索引. 如果没有任何索引可以使用，就会显示成 null，这一项内容对于优化时候索引的调整非常重要；
- ◆ Key：MySQL Query Optimizer 从 possible_keys 中所选择使用的索引；
- ◆ Key_len：被选中使用索引的索引键长度；
- ◆ Ref：列出是通过常量（const），还是某个表的某个字段（如果是 join）来过滤（通过 key）的；
- ◆ Rows：MySQL Query Optimizer 通过系统收集到的统计信息估算出来的结果集记录条数；
- ◆ Extra：查询中每一步实现的额外细节信息，主要可能会是以下内容：
  - ◇ Distinct：查找 distinct 值，所以当 mysql 找到了第一条匹配的结果后，将停止该值的查询而转为后面其他值的查询；
  - ◇ Full scan on NULL key：子查询中的一种优化方式，主要在遇到无法通过索引访问 null 值的使用使用；
  - ◇ Impossible WHERE noticed after reading const tables：MySQL Query Optimizer 通过收集到的统计信息判断出不可能存在结果；
  - ◇ No tables：Query 语句中使用 FROM DUAL 或者不包含任何 FROM 子句；
  - ◇ Not exists：在某些左连接中 MySQL Query Optimizer 所通过改变原有 Query 的组成而使用的优化方法，可以部分减少数据访问次数；
  - ◇ Range checked for each record (index map: N)：通过 MySQL 官方手册的描述，当 MySQL Query Optimizer 没有发现好的可以使用的索引的时候，如果发现如果来自前面的表的列值已知，可能部分索引可以使用。对前面的表的每个行组合，MySQL 检查是否可以使

用 range 或 index_merge 访问方法来索取行。

- ◇ Select tables optimized away：当我们使用某些聚合函数来访问存在索引的某个字段的时候，MySQL Query Optimizer 会通过索引而直接一次定位到所需的数据行完成整个查询。当然，前提是在 Query 中不能有 GROUP BY 操作。如使用 MIN()或者 MAX（）的时候；
- ◇ Using filesort：当我们的 Query 中包含 ORDER BY 操作，而且无法利用索引完成排序操作的时候，MySQL Query Optimizer 不得不选择相应的排序算法来实现。
- ◇ Using index：所需要的数据只需要在 Index 即可全部获得而不需要再到表中取数据；
- ◇ Using index for group-by：数据访问和 Using index 一样，所需数据只需要读取索引即可，而当 Query 中使用了 GROUP BY 或者 DISTINCT 子句的时候，如果分组字段也在索引中，Extra 中的信息就会是 Using index for group-by；
- ◇ Using temporary：当 MySQL 在某些操作中必须使用临时表的时候，在 Extra 信息中就会出现 Using temporary 。主要常见于 GROUP BY 和 ORDER BY 等操作中。
- ◇ Using where：如果我们不是读取表的所有数据，或者不是仅仅通过索引就可以获取所有需要的数据，则会出现 Using where 信息；
- ◇ Using where with pushed condition：这是一个仅仅在 NDBCluster 存储引擎中才会出现的信息，而且还需要通过打开 Condition Pushdown 优化功能才可能会被使用。控制参数为 engine_condition_pushdown 。

这里我们通过分析示例来看一下不同的 Query 语句通过 Explain 所显示的不同信息：

我们先看一个简单的单表 Query：

```
sky@localhost : example 11:33:18> explain select count(*),max(id),min(id)
    -> from user\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Select tables optimized away
```

对 user 表的单表查询，查询类型为 SIMPLE，因为既没有 UNION 也不是子查询。聚合函数 MAX MIN 以及 COUNT 三者所需要的数据都可以通过索引就能够直接定位得到数据，所以整个实现的 Extra 信息为 Select tables optimized away。

再来看一个稍微复杂一点的 Query，一个子查询：

```
sky@localhost : example 11:38:48> explain select name from groups
    -> where id in ( select group_id from user_group where user_id = 1)\G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: PRIMARY
        table: groups
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 50000
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: user_group
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra: Using where
```

通过 id 信息我们可以得知 MySQL Query Optimizer 给出的执行计划是首先对 groups 进行全表扫描，然后第二步才访问 user_group 表，所使用的查询方式是 DEPENDENT SUBQUERY，对所需数据的访问方式是索引扫描，由于过滤条件是一个整数，所以索引扫描的类型为 ref，过滤条件是 const。可以使用的索引有两个，一个是基于 user_id，另一个则是基于 group_id 的。为什么基于 group_id 的索引 user_group_gid_ind 也被列为可选索引了呢？是因为与子查询的外层查询所关联的条件是基于 group_id 的。当然，最后 MySQL Query Optimizer 还是选择了使用基于 user_id 的索引 user_group_uid_ind。

由于篇幅关系，这里就不再继续举例了，大家可以通过自行通过 Explain 功能分析各自应用环境中的各种 Query，了解他们在我们的 MySQL 中到底是怎么运行的。

8.3.2 Profiling 的使用

在本章第一节中我们还提到过通过 Query Profiler 来定位一条 Query 的性能瓶颈，这里我们再详细介绍一下 Profiling 的用途及使用方法。

要想优化一条 Query，我们就需要清楚的知道这条 Query 的性能瓶颈到底在哪里，是消耗的 CPU 计算太多，还是需要的的 IO 操作太多？要想能够清楚的了解这些信息，在 MySQL 5.0 和 MySQL 5.1 正式版中已经可以非常容易做到了，那就是通过 Query Profiler 功能。

MySQL 的 Query Profiler 是一个使用非常方便的 Query 诊断分析工具，通过该工具可以获取一条 Query 在整个执行过程中多种资源的消耗情况，如 CPU，IO，IPC，SWAP 等，以及发生的 PAGE FAULTS，

CONTEXT SWITCHE 等等，同时还能得到该 Query 执行过程中 MySQL 所调用的各个函数在源文件中的位置。下面我们看看 Query Profiler 的具体用法。

1、 开启 profiling 参数

```
root@localhost : (none) 10:53:11> set profiling=1;
Query OK, 0 rows affected (0.00 sec)
```

通过执行 “set profiling”命令，可以开启关闭 Query Profiler 功能。

2、 执行 Query

```
    ... ...
root@localhost : test 07:43:18> select status,count(*)
    -> from test_profiling group by status;
+----------------+----------+
| status         | count(*) |
+----------------+----------+
| st_xxx1        |       27 |
| st_xxx2        |     6666 |
| st_xxx3        |   292887 |
| st_xxx4        |       15 |
+----------------+----------+
5 rows in set (1.11 sec)
    ... ...
```

在开启 Query Profiler 功能之后，MySQL 就会自动记录所有执行的 Query 的 profile 信息了。

3、获取系统中保存的所有 Query 的 profile 概要信息

```
root@localhost : test 07:47:35> show profiles;
+----------+------------+------------------------------------------------------------+
| Query_ID | Duration   | Query                                                      |
+----------+------------+------------------------------------------------------------+
|        1 | 0.00183100 | show databases                                             |
|        2 | 0.00007000 | SELECT DATABASE()                                          |
|        3 | 0.00099300 | desc test                                                  |
|        4 | 0.00048800 | show tables                                                |
|        5 | 0.00430400 | desc test_profiling                                        |
|        6 | 1.90115800 | select status,count(*) from test_profiling group by status |
+----------+------------+------------------------------------------------------------+
3 rows in set (0.00 sec)
```

通过执行 “SHOW PROFILE” 命令获取当前系统中保存的多个 Query 的 profile 的概要信息。

4、针对单个 Query 获取详细的 profile 信息。

在获取到概要信息之后，我们就可以根据概要信息中的 Query_ID 来获取某个 Query 在执行过程中

详细的 profile 信息了，具体操作如下：

root@localhost : test 07:49:24> show profile cpu, block io for query 6;

Status	Duration	CPU_user	CPU_system	Block_ops_in	Block_ops_out
starting	0.000349	0.000000	0.000000	0	0
Opening tables	0.000012	0.000000	0.000000	0	0
System lock	0.000004	0.000000	0.000000	0	0
Table lock	0.000006	0.000000	0.000000	0	0
init	0.000023	0.000000	0.000000	0	0
optimizing	0.000002	0.000000	0.000000	0	0
statistics	0.000007	0.000000	0.000000	0	0
preparing	0.000007	0.000000	0.000000	0	0
Creating tmp table	0.000035	0.000999	0.000000	0	0
executing	0.000002	0.000000	0.000000	0	0
Copying to tmp table	1.900619	1.030844	0.197970	347	347
Sorting result	0.000027	0.000000	0.000000	0	0
Sending data	0.000017	0.000000	0.000000	0	0
end	0.000002	0.000000	0.000000	0	0
removing tmp table	0.000007	0.000000	0.000000	0	0
end	0.000002	0.000000	0.000000	0	0
query end	0.000003	0.000000	0.000000	0	0
freeing items	0.000029	0.000000	0.000000	0	0
logging slow query	0.000001	0.000000	0.000000	0	0
logging slow query	0.000002	0.000000	0.000000	0	0
cleaning up	0.000002	0.000000	0.000000	0	0

上面的例子中是获取 CPU 和 Block IO 的消耗，非常清晰，对于定位性能瓶颈非常适用。希望得到取其他的信息，都可以通过执行 “SHOW PROFILE *** FOR QUERY n” 来获取，各位读者朋友可以自行测试熟悉。

8.4 合理设计并利用索引

索引，可以说是数据库相关优化尤其是在 Query 优化中最常用的优化手段之一了。但是很多人在大部分时候都只是大概了解索引的用途，知道索引能够让 Query 执行的更快，而并不知道为什么会更快。尤其是索引的实现原理，存储方式，以及不同索引之间的区别等就更不是太清楚了。正因为索引对我们的 Query 性能影响很大，所以我们更应该深入理解 MySQL 中索引的基本实现，以及不同索引之间的区别，才能分析出如何设计出最优的索引来最大幅度的提升 Query 的执行效率。

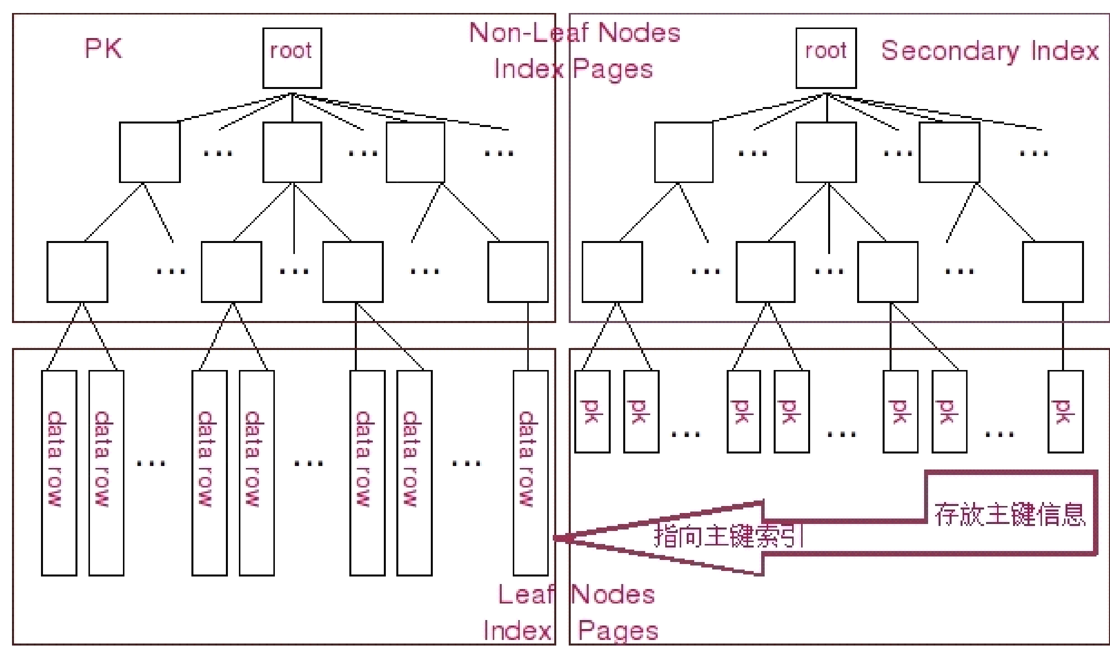
在 MySQL 中，主要有四种类型的索引，分别为：B-Tree 索引，Hash 索引，Fulltext 索引和 R-Tree 索引，下面针对这四种索引的基本实现方式及存储结构做一个大概的分析。

B-Tree 索引

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。不仅仅在 MySQL 中是如此，实际上在其他的很多数据库管理系统中 B-Tree 索引也同样是作为最主要的索引类型，这主要是因为 B-Tree 索引的存储结构在数据库的数据检索中有非常优异的表现。

一般来说，MySQL 中的 B-Tree 索引的物理文件大多都是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以我们大家都称之为 B-Tree 索引当然，可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 Innodb 存储引擎的 B-Tree 索引实际使用的存储结构实际上是 B+Tree，也就是在 B-Tree 数据结构的基础上做了很小的改造，在每一个 Leaf Node 上面出了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 Leaf Node 的指针信息，这主要是为了加快检索多个相邻 Leaf Node 的效率考虑。

在 Innodb 存储引擎中，存在两种不同形式的索引，一种是 Cluster 形式的主键索引（Primary Key），另外一种则是和其他存储引擎（如 MyISAM 存储引擎）存放形式基本相同的普通 B-Tree 索引，这种索引在 Innodb 存储引擎中被称为 Secondary Index。下面我们通过图示来针对这两种索引的存放形式做一个比较。



图示中左边为 Clustered 形式存放的 Primary Key，右侧则为普通的 B-Tree 索引。两种索引在 Root Node 和 Branch Nodes 方面都还是完全一样的。而 Leaf Nodes 就出现差异了。在 Primary Key 中，Leaf Nodes 存放的是表的实际数据，不仅仅包括主键字段的数据，还包括其他字段的数据，整个数据以主键值有序的排列。而 Secondary Index 则和其他普通的 B-Tree 索引没有太大的差异，只是在 Leaf Nodes 出了存放索引键的相关信息外，还存放了 Innodb 的主键值。

所以，在 Innodb 中如果通过主键来访问数据效率是非常高的，而如果是通过 Secondary Index 来访问数据的话，Innodb 首先通过 Secondary Index 的相关信息，通过相应的索引键检索到 Leaf Node 之后，需要再通过 Leaf Node 中存放的主键值再通过主键索引来获取相应的数据行。

MyISAM 存储引擎的主键索引和非主键索引差别很小，只不过是主键索引的索引键是一个唯一且非空的键而已。而且 MyISAM 存储引擎的索引和 Innodb 的 Secondary Index 的存储结构也基本相同，主要的区别只是 MyISAM 存储引擎在 Leaf Nodes 上面出了存放索引键信息之外，再存放能直接定位到 MyISAM 数据文件中相应的数据行的信息（如 Row Number），但并不会存放主键的键值信息。

Hash 索引

Hash 索引在 MySQL 中使用的并不是很多，目前主要是 Memory 存储引擎使用，而且在 Memory 存储引擎中将 Hash 索引作为默认的索引类型。所谓 Hash 索引，实际上就是通过一定的 Hash 算法，将需要索引的键值进行 Hash 运算，然后将得到的 Hash 值存入一个 Hash 表中。然后每次需要检索的时候，都会将检索条件进行相同算法的 Hash 运算，然后再和 Hash 表中的 Hash 值进行比较并得出相应的信息。

在 Memory 存储引擎中，MySQL 还支持非唯一的 Hash 索引。可能很多人会比较惊讶，如果是非唯一的 Hash 索引，那相同的值该如何处理呢？在 Memory 存储引擎的 Hash 索引中，如果遇到非唯一值，存储引擎会将他们链接到同一个 hash 键值下以一个 链表的形式存在，然后在取得实际键值的时候时候再过滤不符合的键。

由于 Hash 索引结构的特殊性，其检索效率非常的高，索引的检索可以一次定位，而不需要像 B-Tree 索引需要从根节点再到枝节点最后才能访问到页节点这样多次 IO 访问，所以 Hash 索引的效率要远高于 B-Tree 索引。

可能很多人又会有疑问了，既然 Hash 索引的效率要比 B-Tree 高很多，为什么大家不都用 Hash 索引而还要使用 B-Tree 索引呢？任何事物都是有两面性的，，Hash 索引也一样，虽然 Hash 索引检索效率非常之高，但是 Hash 索引本身由于其实的特殊性也带来了很多限制和弊端，主要有以下这些：

1. Hash 索引仅仅只能满足“=”,“IN”和“<=>”查询，不能使用范围查询；
   由于 Hash 索引所比较的是进行 Hash 运算之后的 Hash 值，所以 Hash 索引只能用于等值的过滤，而不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证还和 Hash 运算之前完全一样。
2. Hash 索引无法被利用来避免数据的排序操作；
   由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值的完全一样，所以数据库无法利用索引的数据来避免任何和排序运算；
3. Hash 索引不能利用部分索引键查询；
   对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并之后再一起计算 Hash 值，而不是单独计算 Hash 值，所以当我们通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用到；
4. Hash 索引在任何时候都不能避免表扫面；
   前面我们已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，而且由于存在不同索引键存在相同 Hash 值的

可能，所以即使我们仅仅取满足某个 Hash 键值的数据的记录条数，都无法直接从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较而得到相应的结果。

5. Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高；
对于选择性比较低的索引键，如果我们创建 Hash 索引，那么我们将会存在大量记录指针信息存与同一个 Hash 值相关连。这样要定位某一条记录的时候就会非常的麻烦，可能会浪费非常多次表数据的访问，而造成整体性能的地下。

Full-text 索引

Full-text 索引也就是我们常说的全文索引，目前在 MySQL 中仅有 MyISAM 存储引擎支持，而且也并不是所有的数据类型都支持全文索引。目前来说，仅有CHAR，VARCHAR和TEXT这三种数据类型的列可以建 Full-text 索引。

一般来说，Fulltext 索引主要用来替代效率低下的 LIKE '%***%' 操作。实际上，Full-text 索引并不只是能简单的替代传统的全模糊 LIKE 操作，而且能通过多字段组合的 Full-text 索引一次全模糊匹配多个字段。

Full-text 索引和普通的 B-Tree 索引的实现区别较大，虽然他同样是以 B-Tree 形式来存放索引数据，但是他并不是通过字段内容的完整匹配，而是通过特定的算法，将字段数据进行分隔后再进行的索引。一般来说 MySQL 系统会按照四个字节来分隔。在整个 Full-text 索引中，存储内容被分为两部分，一部分是分隔前的索引字符串数据集合，另一部分是分隔后的词（或者词组）的索引信息。所以，Full-text 索引中，真正在 B-Tree 索引细细中的并不是我们表中的原始数据，而是分词之后的索引数据。在 B-Tree 索引的节点信息中，存放了各个分隔后的词信息，以及指向包含该词的分隔前字符串信息在索引数据集合中的位置信息。

Full-text 索引不仅仅能实现模糊匹配查找，在实现了基于自然语言的的匹配度查找。当然，这个匹配读到底有多准确就需要读者朋友去自行验证了。Full-text 通过一些特定的语法信息，针对自然语言做了各种相应规则的匹配，最后给出非负的匹配值。

此外，有一点是需要大家注意的，MySQL 目前的 Full-text 索引在中文支持方面还不太好，需要借助第三方的补丁或者插件来完成。而且 Full-text 的创建所消耗的资源也是比较大的，所以在应用于实际生产环境之前还是尽量做好评估。

关于 Full-text 的实际使用方法由于不是本书的重点，感兴趣的读者朋友可以自行参阅 MySQL 关于 Full-text 相关的使用手册来了解更为详尽的信息。

R-Tree 索引

R-Tree 索引可能是我们在其他数据库中很少见到的一种索引类型，主要用来解决空间数据检索的问题。

在 MySQL 中，支持一种用来存放空间信息的数据类型GEOMETRY，且基于 OpenGIS 规范。在 MySQL5.0.16 之前的版本中，仅仅 MyISAM 存储引擎支持该数据类型，但是从 MySQL5.0.16 版本开始，

BDB，Innodb，NDBCluster 和 Archive 存储引擎也开始支持该数据类型。当然，虽然多种存储引擎都开始支持 GEOMETRY 数据类型，但是仅仅之后 MyISAM 存储引擎支持 R-Tree 索引。

在 MySQL 中采用了具有二次分裂特性的 R-Tree 来索引空间数据信息，然后通过几何对象（MRB）信息来创建索引。

虽然仅仅只有 MyISAM 存储引擎支持空间索引（R-Tree Index），但是如果我们是精确的等值匹配，创建在空间数据上面的 B-Tree 索引同样可以起到优化检索的效果，空间索引的主要优势在于当我们使用范围查找的时候，可以利用到 R-Tree 索引，而这时候，B-Tree 索引就无能为力了。

对于 R-Tree 索引的详细介绍和使用信息清参阅 MySQL 使用手册。

索引的利弊与如何判定是否需要索引

相信没一位读者朋友都知道索引能够极大的提高我们数据检索的效率，让我们的 Query 执行的更快，但是可能并不是每一位朋友都清楚索引在极大提高检索效率的同时，也给我们的数据库带来了一些负面的影响。下面我们就分别对 MySQL 中索引的利与弊做一个简单的分析。

索引的利处

索引能够给我们带来的最大益处可能读者朋友基本上都有一定的了解，但是我相信并不是每一位读者朋友都能够了解的比较全面。很多朋友对数据库中的索引的认识可能主要还是只限于“能够提高数据检索的效率，降低数据库的 IO 成本”。

确实，在数据库中个表的某个字段创建索引，所带来的最大益处就是将该字段作为检索条件的时候可以极大的提高检索效率，加快检索时间，降低检索过程中所需要读取的数据量。但是索引所给我们带来的收益只是提高表数据的检索效率吗？当然不是，索引还有一个非常重要的用途，那就是降低数据的排序成本。

我们知道，每个索引中索引数据都是按照索引键键值进行排序后存放的，所以，当我们的 Query 语句中包含排序分组操作的时候，如果我们的排序字段和索引键字段刚好一致，MySQL Query Optimizer 就会告诉 mysqld 在取得数据之后不用排序了，因为根据索引取得的数据已经是满足客户的排序要求。

那如果是分组操作呢？分组操作没办法直接利用索引完成。但是分组操作是需要先进行排序然后才分组的，所以当我们的 Query 语句中包含分组操作，而且分组字段也刚好和索引键字段一致，那么 mysqld 同样可以利用到索引已经排好序的这个特性而省略掉分组中的排序操作。

排序分组操作主要消耗的是我们的内存和 CPU 资源，如果我们能够在进行排序分组操作中利用好索引，将会极大的降低 CPU 资源的消耗。

索引的弊端

索引的益处我们都已经清楚了，但是我们不能光看到索引给我们带来的这么多益处之后就认为索引是解决 Query 优化的圣经，只要发现 Query 运行不够快就将 WHERE 子句中的条件全部放在索引中。

确实，索引能够极大的提高数据检索效率，也能够改善排序分组操作的性能，但是我们不能忽略的一个问题就是索引是完全独立于基础数据之外的一部分数据。假设我们在 Table ta 中的 Column ca 创建了索引 idx_ta_ca，那么任何更新 Column ca 的操作，MySQL 都需要在更新表中 Column ca的同时，也更新 Column ca 的索引数据，调整因为更新所带来键值变化后的索引信息。而如果我们没有对Column ca 进行索引的话,MySQL 所需要做的仅仅只是更新表中 Column ca 的信息。这样，所带来的最明显的资源消耗就是增加了更新所带来的 IO 量和调整索引所致的计算量。此外，Column ca 的索引idx_ta_ca 是需要占用存储空间的，而且随着 Table ta 数据量的增长，idx_ta_ca 所占用的空间也会不断增长。所以索引还会带来存储空间资源消耗的增长。

如何判定是否需要创建索引

在了解了索引的利与弊之后，我们知道了索引并不是越多越好，知道了索引也是会带来副作用的。那我们到底该如何来判断某个索引是否应该创建呢？

实际上，并没有一个非常明确的定律可以清晰的定义出什么字段应该创建索引什么字段不该创建索引。因为我们的应用场景实在是太复杂，存在太多的差异。当然，我们还是仍然能够找到几点基本的判定策略来帮助我们分析是否需要创建索引。

◆ 较频繁的作为查询条件的字段应该创建索引；
  提高数据查询检索的效率最有效的办法就是减少需要访问的数据量，从上面所了解到的索引的益处中我们知道了，索引正是我们减少通过索引键字段作为查询条件的 Query 的 IO 量的最有效手段。所以一般来说我们应该为较为频繁的查询条件字段创建索引。

◆ 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件；
  唯一性太差的字段主要是指哪些呢？如状态字段，类型字段等等这些字段中存方的数据可能总共就是那么几个几十个值重复使用，每个值都会存在于成千上万或是更多的记录中。对于这类字段，我们完全没有必要创建单独的索引的。因为即使我们创建了索引，MySQL Query Optimizer 大多数时候也不会去选择使用，如果什么时候 MySQL Query Optimizer 抽了一下风选择了这种索引，那么非常遗憾的告诉你，这可能会带来极大的性能问题。由于索引字段中每个值都含有大量的记录，那么存储引擎在根据索引访问数据的时候会带来大量的随机IO，甚至有些时候可能还会出现大量的重复IO。

  这主要是由于数据基于索引扫描的特点所引起的。当我们通过索引访问表中的数据的时候，MySQL 会按照索引键的键值的顺序来依序进行访问。一般来说每个数据页中大都会存放多条记录，但是这些记录可能大多数都不会是和你所使用的索引键的键值顺序一致。

  假如有以下场景，我们通过索引查找键值为 A 和 B 的某些数据。当我们先通过 A 键值找到第一条满足要求的记录后，我们会读取这条记录所在的 X 数据页，然后我们继续往下查找索引，发现 A 键值所对应的另外一条记录也满足我们的要求，但是这条记录不在 X 数据页上面，而在 Y 数据页上面，这时候存储引擎就会丢弃 X 数据页，而读取 Y 数据页。如此继续一直到查找完 A 键值所对应的所有记录。然后轮到 B 键值了，这时候发现正在查找的记录又在 X 数据页上面，可之前读取的 X 数据页已经被丢弃了，只能再次读取 X 数据页。这时候，实际上已经出现重复读取 X 数据页两次了。在继续往后的查找中，可能还会出现一次又一次的重复读取。

这无疑极大的给存储引擎增大了 IO 访问量。

不仅如此，如果一个键值对应了太多的数据记录，也就是说通过该键值会返回占整个表比例很大的记录的时候，由于根据索引扫描产生的都是随机 IO，其效率比进行全表扫描的顺序 IO 的效率要差很多，即使不会出现重复 IO 的读取，同样会造成整体 IO 性能的下降。

很多比较有经验的 Query 调优专家经常说，当一条 Query 所返回的数据超过了全表的 15% 的时候，就不应该再使用索引扫描来完成这个 Query 了。对于“15%”这个数字我们并不能判定是否很准确，但是之少侧面证明了唯一性太差的字段并不适合创建索引。

◆ 更新非常频繁的字段不适合创建索引；

上面在索引的弊端中我们已经分析过了，索引中的字段被更新的时候，不仅仅需要更新表中的数据，同时还要更新索引数据，以确保索引信息是准确的。这个问题所带来的是 IO 访问量的较大增加，不仅仅影响更新 Query 的响应时间，还会影响整个存储系统的资源消耗，加大整个存储系统的负载。

当然，并不是存在更新的字段就比适合创建索引，从上面判定策略的用语上面也可以看出，是“非常频繁”的字段。到底什么样的更新频率应该算是“非常频繁”呢？每秒，每分钟，还是每小时呢？说实话，这个还真挺难定义的。很多时候还是通过比较同一时间段内被更新的次数和利用该字段作为条件的查询次数来判断，如果通过该字段的查询并不是很多，可能几个小时或者是更长才会执行一次，而更新反而比查询更频繁，那这样的字段肯定不适合创建索引。反之，如果我们通过该字段的查询比较频繁，而且更新并不是特别多，比如查询十几二十次或是更多才可能会产生一次更新，那我个人觉得更新所带来的附加成本也是可以接受的。

◆ 不会出现在 WHERE 子句中的字段不该创建索引；

不会还有人会问为什么吧？自己也觉得这是废话了，哈哈！

单键索引还是组合索引

在大概了解了一下 MySQL 各种类型的索引以及索引本身的利弊与判断一个字段是否需要创建索引之后，我们就需要着手创建索引来优化我们的 Query 了。在很多时候，我们的 WHERE 子句中的过滤条件并不只是针对于单一的某个字段，而是经常会有多个字段一起作为查询过滤条件存在于 WHERE 子句中。在这种时候，我们就必须要作出判断，是该仅仅为过滤性最好的字段建立索引还是该在所有字段（过滤条件中的）上面建立一个组合索引呢？

对于这种问题，很难有一个绝对的定论，我们需要从多方面来分析考虑，平衡两种方案各自的优劣，然后选择一种最佳的方案来解决。因为从上一节中我们了解到了索引在提高某些查询的性能的同时，也会让某些更新的效率下降。而组合索引中因为有多个字段的存在，理论上被更新的可能性肯定比单键索引要大很多，这样可能带来的附加成本也就比单键索引要高。但是，当我们的 WHERE 子句中的查询条件含有多个字段的时候，通过这多个字段共同组成的组合索引的查询效率肯定比仅仅只用过滤条件中的某一个字段创建的索引要高。因为通过单键索引所能过滤的数据并不完整，和通过组合索引相比，存储引擎需要访问更多的记录数，自然就会访问更多的数据量，也就是说需要更高的 IO 成本。

可能有些朋友会说，那我们可以通过创建多个单键索引啊。确实，我们可以将 WHERE 子句中的每一个字段都创建一个单键索引。但是这样真的有效吗？在这样的情况下，MySQL Query Optimizer 大多数时候都只会选择其中的一个索引，然后放弃其他的索引。即使他选择了同时利用两个或者更多的索引通过 INDEX_MERGE 来优化查询，可能所收到的效果并不会比选择其中某一个单键索引更高效。因为如果选择通过 INDEX_MERGE 来优化查询，就需要访问多个索引，同时还要将通过访问到的几个索引进行 merge 操作，所带来的成本可能反而会比选择其中一个最有效的索引来完成查询更高。

在一般的应用场景中，只要不是其中某个过滤字段在大多数场景下都能过滤出 90%以上的数据，而且其他的过滤字段会存在频繁的更新，我一般更倾向于创建组合索引，尤其是在并发量较高的场景下更是应该如此。因为当我们的并发量较高的时候，即使我们为每个 Query 节省很少的 IO 消耗，但因为执行量非常大，所节省的资源总量仍然是非常可观的。

当然，我们创建组合索引并不是说就需要将查询条件中的所有字段都放在一个索引中，我们还应该尽量让一个索引被多个 Query 语句所利用，尽量减少同一个表上面索引的数量，减少因为数据更新所带来的索引更新成本，同时还可以减少因为索引所消耗的存储空间。

此外，MySQL 还为我们提供了一个减少优化索引自身的功能，那就是前缀索引。在 MySQL 中，我们可以仅仅使用某个字段的前面部分内容做为索引键来索引该字段，来达到减小索引占用的存储空间和提高索引访问的效率。当然，前缀索引的功能仅仅适用于字段前缀比较随机重复性很小的字段。如果我们需要索引的字段的前缀内容有较多的重复，索引的过滤性自然也会随之降低，通过索引所访问的数据量就会增加，这时候前缀索引虽然能够减少存储空间消耗，但是可能会造成 Query 访问效率的极大降低，反而得不偿失。

Query 的索引选择

在有些场景下，我们的 Query 由于存在多个过滤条件，而这多个过滤条件可能会存在于两个或者更多的索引中。在这种场景下，MySQL Query Optimizer 一般情况下都能够根据系统的统计信息选择出一个针对该 Query 最优的索引完成查询，但是在有些情况下，可能是由于我们的系统统计信息的不够准确完整，也可能是 MySQL Query Optimizer 自身功能的缺陷，会造成他并没有选择一个真正最优的索引而选择了其他查询效率较低的索引。在这种时候，我们就不得不通过认为干预，在 Query 中增加 Hint 提示 MySQL Query Optimizer 告诉他该使用哪个索引而不该使用哪个索引，或者通过调整查询条件来达到相同的目的。

我们这里再次通过在本章第 2 节“Query 语句优化基本思路和原则”的“仅仅使用最有效的过滤条件”中示例的基础上将 group_message 表的索引做部分调整，然后再进行分析。

在 group_message 上增加如下索引：

```
create index group_message_author_subject on group_message(author,subject(16));
```

调整后的索引信息如下（出于篇幅考虑省略了主键索引）：

```
sky@localhost : example 07:13:38> show indexes from group_message\G
    ......
*************************** 2. row ***************************
        Table: group_message
```

```
   Non_unique: 1
     Key_name: group_message_author_subject
 Seq_in_index: 1
  Column_name: author
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 3. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: group_message_author_subject
 Seq_in_index: 2
  Column_name: subject
    Collation: A
  Cardinality: NULL
     Sub_part: 16
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 4. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: idx_group_message_uid
 Seq_in_index: 1
  Column_name: user_id
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 5. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: idx_group_message_author
 Seq_in_index: 1
  Column_name: author
    Collation: A
```

```
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

从索引的 Sub_part 中，我们可以看到 subject 字段是取前 16 个字符的前缀作为索引键。下面假设我们知道某个用户的 user_id ，nick_name 和 subject 字段的部分前缀信息（weiurazs），希望通过这些条件查询出所有满足上面存在于 group_message 中的信息。我们知道存在三个索引可以被利用：idx_group_message_author ， idx_group_message_uid 和 group_message_author_subject，而且也知道每个 user_id 实际上都是和 一个 author 分别唯一对应的。所以实际上，无论是使用 user_id 和 author（nick_name）中的某一个来作为条件或者两个条件都使用，所得到的数据都是完全一样的。当然，我们还需要 subject LIKE 'weiurazs%' 这个条件来过滤 subject 相关的信息。

根据三个索引的组成，和我们的查询条件，我们知道 group_message_author_subject 索引可以让我们得到最高的检索效率，因为只有他索引了 subject 相关的信息，subject 是我们的查询必须包含的过滤条件。下面我们分别看看使用 user_id ，author 和 两者共同使用时候的执行计划。

```
sky@localhost : example 07:48:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_group_message_uid
          key: idx_group_message_uid
      key_len: 4
          ref: const
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

很明显，这不是我们所期望的执行计划，当然我们并不能责怪 MySQL，因为我们都没有使用 author 来进行过滤，Optimizer 当然不会选择 group_message_author_subject 这个索引，这是我们自己的错。

```
sky@localhost : example 07:48:49> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
```

```
         type: range
possible_keys: group_message_author_subject,idx_group_message_author
          key: idx_group_message_author
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

这次我们改为使用 author 作为查询条件了，可 MySQL Query Optimizer 仍然没有选择 group_message_author_subject 这个索引，即使我们通过 analyze 分析也是同样的结果。

```
sky@localhost : example 07:48:57> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject,idx_group_message_uid,
               idx_group_message_author
          key: idx_group_message_uid
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

同时使用 user_id 和 author 两者的时候，MySQL Query Optimizer 又再次选择了 idx_group_message_uid 这个索引，仍然不是我们期望的结果。

```
sky@localhost : example 07:51:11>  EXPLAIN SELECT * FROM group_message
    -> FORCE INDEX(idx_group_message_author_subject)
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject
          key: group_message_author_subject
      key_len: 148
          ref: NULL
         rows: 8
```

```
Extra: Using where
```

在最后，我们不得不利用 MySQL 为我们提供的在优化 Query 时候所使用的高级功能，通过显式告诉 MySQL Query Optimizer 我们要使用哪个索引的 Hint 功能。强制 MySQL 使用 group_message_author_subject 这个索引来完成查询，才达到我们所需要的效果。

或许有些读者会想，会不会是因为选择 group_message_author_subject 这个索引本身就不是一个最有的选择呢？大家请看下面通过 mysqlslap 进行的实际执行各条 Query 的测试结果：

```
sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE
user_id = 3 AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.021 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE
author = '3' AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.025 seconds
    Minimum number of seconds to run all queries: 0.012 seconds
    Maximum number of seconds to run all queries: 0.031 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE
user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.026 seconds
    Minimum number of seconds to run all queries: 0.013 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message force
index(group_message_author_subject) WHERE author = '3' subject LIKE 'weiurazs%'" --
iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.017 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.027 seconds
    Number of clients running queries: 1
```

Average number of queries per client: 1

我们可以清晰的看出，通过我们添加 Hint 之后选择 group_message_author_subject 这个索引的 Query 确实比其他的三条要快很多。

通过这个示例，我们可以看出在优化 Query 的时候，选择合适的索引是非常重要的，而且我们也同时实例证明了 MySQL Query Optimizer 并不是任何时候都能够选择出最佳的执行计划，在有些时候，我们不得不通过人为的手工干预来让 MySQL Query Optimizer 改变他的“想法”，而按照我们的思路走。

当然，这个示例仅仅只是告诉了我们选择合适索引的重要性，并且不能任何时候都完全相信 MySQL Query Optimizer，但并没有告诉我们到底该如何来选择一个更合适的索引。下面是我对于选择合适索引的几点建议，并不一定在任何场景下都合适，但在大多数场景下还是比较适用的。

1. 对于单键索引，尽量选择针对当前 Query 过滤性更好的索引；
2. 在选择组合索引的时候，当前 Query 中过滤性最好的字段在索引字段顺序中排列越靠前越好；
3. 在选择组合索引的时候，尽量选择可以能够包含当前 Query 的 WHERE 子句中更多字段的索引；
4. 尽可能通过分析统计信息和调整 Query 的写法来达到选择合适索引的目的而减少通过使用 Hint 人为控制索引的选择，因为这会使后期的维护成本增加，同时增加维护所带来的潜在风险。

MySQL 中索引的限制

在使用索引的同时，我们还应该了解在 MySQL 中索引存在的限制，以便在索引应用中尽可能的避开限制所带来的问题。下面列出了目前 MySQL 中索引使用相关的限制。

1. MyISAM 存储引擎索引键长度总和不能超过 1000 字节；
2. BLOB 和 TEXT 类型的列只能创建前缀索引；
3. MySQL 目前不支持函数索引；
4. 使用不等于（!= 或者 <>）的时候 MySQL 无法使用索引；
5. 过滤字段使用了函数运算后（如 abs(column)），MySQL 无法使用索引；
6. Join 语句中 Join 条件字段类型不一致的时候 MySQL 无法使用索引；
7. 使用 LIKE 操作的时候如果条件以通配符开始（ '%abc...'）MySQL 无法使用索引；
8. 使用非等值查询的时候 MySQL 无法使用 Hash 索引；
9.

在我们使用索引的时候，需要注意上面的这些限制，尤其是要注意无法使用索引的情况，因为这很容易让我们因为疏忽而造成极大的性能隐患。

8.5 Join 的实现原理及优化思路

前面我们已经了解了 MySQL Query Optimizer 的工作原理，学习了 Query 优化的基本原则和思路，理解了索引选择的技巧，这一节我们将围绕 Query 语句中使用非常频繁，且随时可能存在性能隐患的 Join 语句，继续我们的 Query 优化之旅。

Join 的实现原理

在寻找 Join 语句的优化思路之前，我们首先要理解在 MySQL 中是如何来实现 Join 的，只要理解了实现原理之后，优化就比较简单了。下面我们先分析一下 MySQL 中 Join 的实现原理。

在 MySQL 中，只有一种 Join 算法，就是大名鼎鼎的 Nested Loop Join，他没有其他很多数据库所提供的 Hash Join，也没有 Sort Merge Join。顾名思义，Nested Loop Join 实际上就是通过驱动表的结果集作为循环基础数据，然后一条一条的通过该结果集中的数据作为过滤条件到下一个表中查询数据，然后合并结果。如果还有第三个参与 Join，则再通过前两个表的 Join 结果集作为循环基础数据，再一次通过循环查询条件到第三个表中查询数据，如此往复。

下面我们将通过一个三表 Join 语句示例来说明 MySQL 的 Nested Loop Join 实现方式。

注意：由于要展示 Explain 中的一个在 MySQL 5.1.18 才开始出现的输出信息（在之前版本中只是没有输出信息，实际执行过程并没有变化），所以下面的示例环境是 MySQL5.1.26。

Query 如下：

```
select m.subject msg_subject, c.content msg_content
from user_group g,group_message m,group_message_content c
where g.user_id = 1
    and m.group_id = g.group_id
    and c.group_msg_id = m.id
```

为了便于示例，我们通过如下操作为 group_message 表增加了一个 group_id 的索引：

```
create index idx_group_message_gid_uid on group_message(group_id);
```

然后看看我们的 Query 的执行计划：

```
sky@localhost : example 11:17:04> explain select m.subject msg_subject, c.content
msg_content
    -> from user_group g,group_message m,group_message_content c
    -> where g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind,user_group_gid_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
```

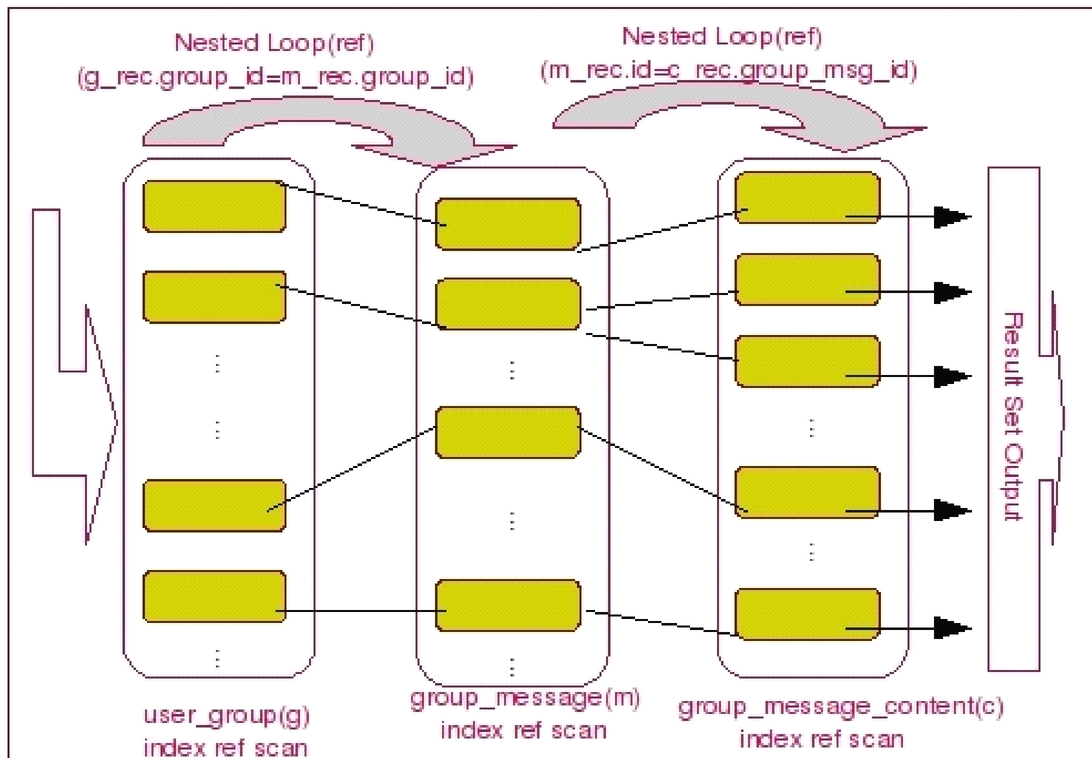
```
             rows: 2
            Extra:
*************************** 2. row ***************************
               id: 1
      select_type: SIMPLE
            table: m
             type: ref
    possible_keys: PRIMARY,idx_group_message_gid_uid
              key: idx_group_message_gid_uid
          key_len: 4
              ref: example.g.group_id
             rows: 3
            Extra:
*************************** 3. row ***************************
               id: 1
      select_type: SIMPLE
            table: c
             type: ref
    possible_keys: idx_group_message_content_msg_id
              key: idx_group_message_content_msg_id
          key_len: 4
              ref: example.m.id
             rows: 2
            Extra:
```

我们可以看出，MySQL Query Optimizer 选择了 user_group 作为驱动表，首先利用我们传入的条件 user_id 通过 该表上面的索引 user_group_uid_ind 来进行 const 条件的索引 ref 查找，然后以 user_group 表中过滤出来的结果集的 group_id 字段作为查询条件，对 group_message 循环查询，然后再通过 user_group 和 group_message 两个表的结果集中的 group_message 的 id 作为条件 与 group_message_content 的 group_msg_id 比较进行循环查询，才得到最终的结果。

这个过程可以通过如下表达式来表示：

```
for each record g_rec in table user_group that g_rec.user_id=1{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  for each record c_rec in group_message_content that c_rec.group_msg_id=m_rec.id
   pass the (g_rec.user_id, m_rec.subject, c_rec.content) row
   combination to output;
 }
}
```

下图可以更清晰的标识出实际的执行情况：



假设我们去掉 group_message_content 表上面的 group_msg_id 字段的索引，然后再看看执行计划会变成怎样：

```
sky@localhost : example 11:25:36> drop index idx_group_message_content_msg_id on
group_message_content;
Query OK, 96 rows affected (0.11 sec)

sky@localhost : example 10:21:06> explain
    -> select m.subject msg_subject, c.content msg_content
    -> from user_group g,group_message m,group_message_content c
    -> where g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: idx_user_group_uid
          key: idx_user_group_uid
      key_len: 4
          ref: const
         rows: 2
        Extra:
```

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: example.g.group_id
         rows: 3
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 96
        Extra: Using where; Using join buffer
```

我们看到不仅仅 user_group 表的访问从 ref 变成了 ALL，此外，在最后一行的 Extra 信息从没有任何内容变成为 Using where; Using join buffer，也就是说，对于从 ref 变成 ALL 很容易理解，没有可以使用的索引的索引了嘛，当然得进行全表扫描了，Using where 也是因为变成全表扫描之后，我们需要取得的 content 字段只能通过对表中的数据进行 where 过滤才能取得，但是后面出现的 Using join buffer 是一个啥呢？

实际上，这里的 Join 正是利用到了我们在之前 “MySQL Server 性能优化”一章中所提到的一个 Cache 参数相关的内容，也就是我们通过 join_buffer_size 参数所设置的 Join Buffer。

实际上，Join Buffer 只有当我们的 Join 类型为 ALL（如示例中），index，rang 或者是 index_merge 的时候 才能够使用，所以，在我们去掉 group_message_content 表的 group_msg_id 字段的索引之前，由于 Join 是 ref 类型的，所以我们的执行计划中并没有看到有使用 Join Buffer。

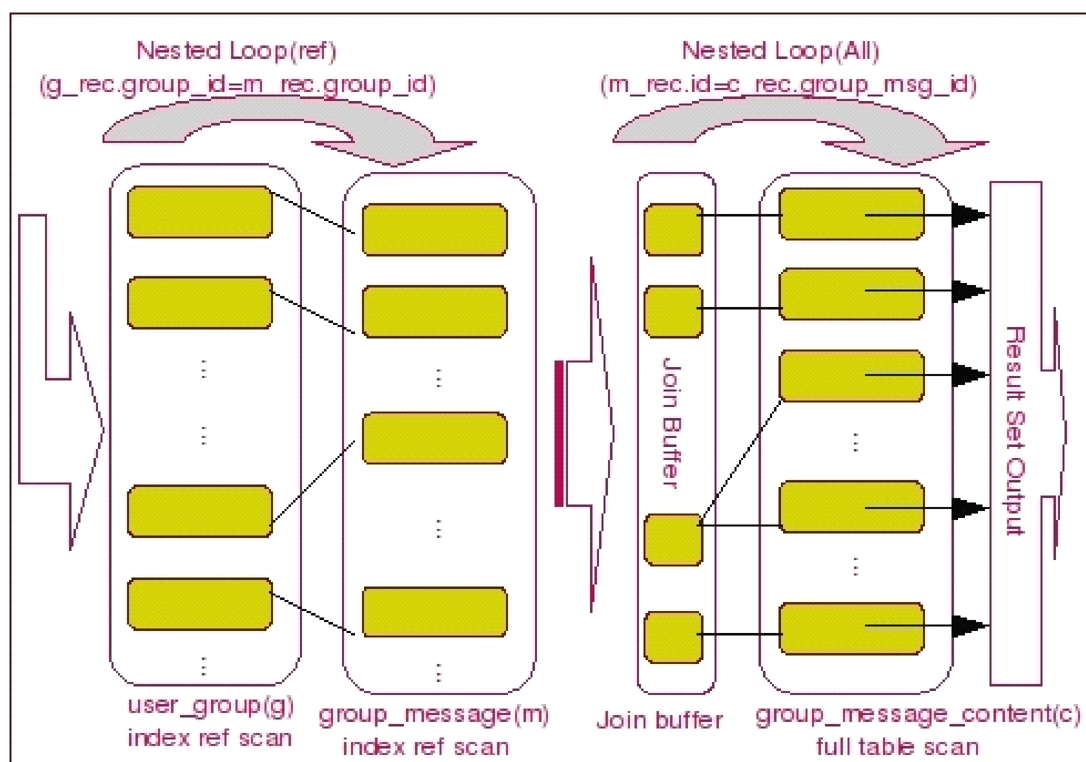
当我们使用了 Join Buffer 之后，我们可以通过下面的这个表达式描述出示例中我们的 Join 完成过程：

```
for each record g_rec in table user_group{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  put (g_rec, m_rec) into the buffer
   if (buffer is full)
     flush_buffer();
```

```
 }
}

flush_buffer(){
 for each record c_rec in group_message_content that
 c_rec.group_msg_id = c_rec.id{
  for each record in the buffer
   pass (g_rec.user_id, m_rec.subject, c_rec.content) row combination to output;
 }
 empty the buffer;
}
```

当然，如果通过类似于上面的图片来展现或许大家会觉得更容易理解一些，如下：



通过上面的示例，我想大家应该对 MySQL 中 Nested Join 的实现原理有了一个了解了，也应该清楚 MySQL 使用 Join Buffer 的方法了。当然，这里并没有涉及到 外连接的内容，实际对于外连接来说，可能存在的区别主要是连接顺序以及组合空值记录方面。

Join 语句的优化

在明白了 MySQL 中 Join 的实现原理之后，我们就比较清楚的知道该如何去优化一个一个 Join 语句了。

1. 尽可能减少 Join 语句中的 Nested Loop 的循环总次数；
   如何减少 Nested Loop 的循环总次数？最有效的办法只有一个，那就是让驱动表的结果集尽可

能的小，这也正是在本章第二节中的优化基本原则之一“永远用小结果集驱动大的结果集”。

为什么？因为驱动结果集越大，意味着需要循环的次数越多，也就是说在被驱动结果集上面所需要执行的查询检索次数会越多。比如，当两个表（表 A 和 表 B） Join 的时候，如果表 A 通过 WHERE 条件过滤后有 10 条记录，而表 B 有 20 条记录。如果我们选择表 A 作为驱动表，也就是被驱动表的结果集为 20，那么我们通过 Join 条件对被驱动表（表 B）的比较过滤就会有 10 次。反之，如果我们选择表 B 作为驱动表，则需要有 20 次对表 A 的比较过滤。

当然，此优化的前提条件是通过 Join 条件对各个表的每次访问的资源消耗差别不是太大。如果访问存在较大的差别的时候（一般都是因为索引的区别），我们就不能简单的通过结果集的大小来判断需要 Join 语句的驱动顺序，而是要通过比较循环次数和每次循环所需要的消耗的乘积的大小来得到如何驱动更优化。

2. 优先优化 Nested Loop 的内层循环；

不仅仅是在数据库的 Join 中应该做的，实际上在我们优化程序语言的时候也有类似的优化原则。内层循环是循环中执行次数最多的，每次循环节约很小的资源，在整个循环中就能节约很大的资源。

3. 保证 Join 语句中被驱动表上 Join 条件字段已经被索引；

保证被驱动表上 Join 条件字段已经被索引的目的，正是针对上面两点的考虑，只有让被驱动表的 Join 条件字段被索引了，才能保证循环中每次查询都能够消耗较少的资源，这也正是优化内层循环的实际优化方法。

4. 当无法保证被驱动表的 Join 条件字段被索引且内存资源充足的前提下，不要太吝惜 Join Buffer 的设置；

当在某些特殊的环境中，我们的 Join 必须是 All，Index，range 或者是 index_merge 类型的时候，Join Buffer 就会派上用场了。在这种情况下，Join Buffer 的大小将对整个 Join 语句的消耗起到非常关键的作用。

8.6 ORDER BY，GROUP BY 和 DISTINCT 优化

除了常规的 Join 语句之外，还有一类 Query 语句也是使用比较频繁的，那就是 ORDER BY，GROUP BY 以及 DISTINCT 这三类查询。考虑到这三类查询都涉及到数据的排序等操作，所以我将他们放在了一起，下面就针对这三类 Query 语句做基本的分析。

ORDER BY 的实现与优化

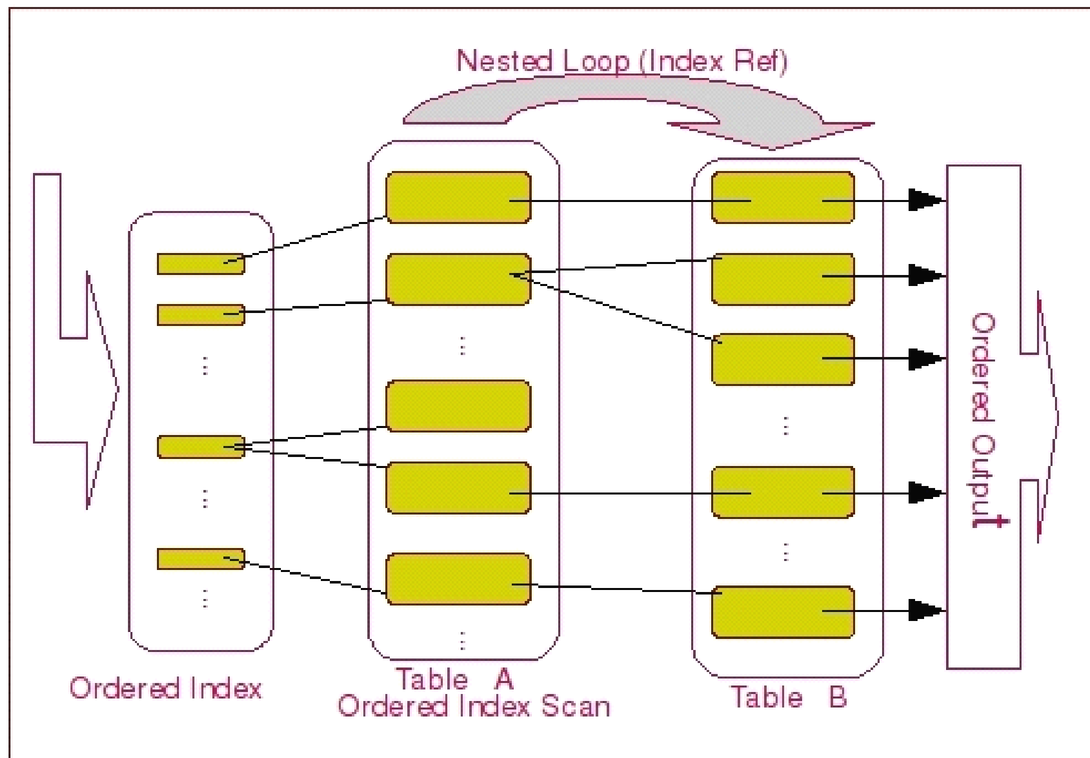
在 MySQL 中，ORDER BY 的实现有如下两种类型：

- 一种是通过有序索引而直接取得有序的数据，这样不用进行任何排序操作即可得到满足客户端要求的有序数据返回给客户端；
- 另外一种则需要通过 MySQL 的排序算法将存储引擎中返回的数据进行排序然后再将排序后的数据返回给客户端。

下面我们就针对这两种实现方式做一个简单的分析。首先分析一下第一种不用排序的实现方式。同样还是通过示例来说话吧：

```
sky@localhost : example 09:48:41> EXPLAIN
    -> SELECT m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: const
         rows: 4
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: group_message_content_msg_id
          key: group_message_content_msg_id
      key_len: 4
          ref: example.m.id
         rows: 11
        Extra:
```

看看上面的这个 Query 语句，明明有 ORDER BY user_id，为什么在执行计划中却没有排序操作呢？其实这里正是因为 MySQL Query Optimizer 选择了一个有序的索引来进行访问表中的数据（idx_group_message_gid_uid），这样，我们通过 group_id 的条件得到的数据已经是按照 group_id 和 user_id 进行排序的了。而虽然我们的排序条件仅仅只有一个user_id，但是我们的 WHERE 条件决定了返回数据的 group_id 全部一样，也就是说不管有没有根据 group_id 来进行排序，返回的结果集都是完全一样的。我们可以通过如下的图示来描述整个执行过程：



图中的 Table A 和 Table B 分别为上面 Query 中的 group_message 和 gruop_message_content 这两个表。

这种利用索引实现数据排序的方法是 MySQL 中实现结果集排序的最佳做法，可以完全避免因为排序计算所带来的资源消耗。所以，在我们优化 Query 语句中的 ORDER BY 的时候，尽可能利用已有的索引来避免实际的排序计算，可以很大幅度的提升 ORDER BY 操作的性能。在有些 Query 的优化过程中，即使为了避免实际的排序操作而调整索引字段的顺序，甚至是增加索引字段也是值得的。当然，在调整索引之前，同时还需要评估调整该索引对其他 Query 所带来的影响，平衡整体得失。

如果没有索引利用的时候，MySQL 又如何来实现排序呢？这时候 MySQL 无法避免需要通过相关的排序算法来将存储引擎返回的数据进行排序运算了。下面我们再针对这种实现方式进行相应的分析。

在 MySQL 第二种排序实现方式中，必须进行相应的排序算法来实现数据的排序。MySQL 目前可以通过两种算法来实现数据的排序操作。

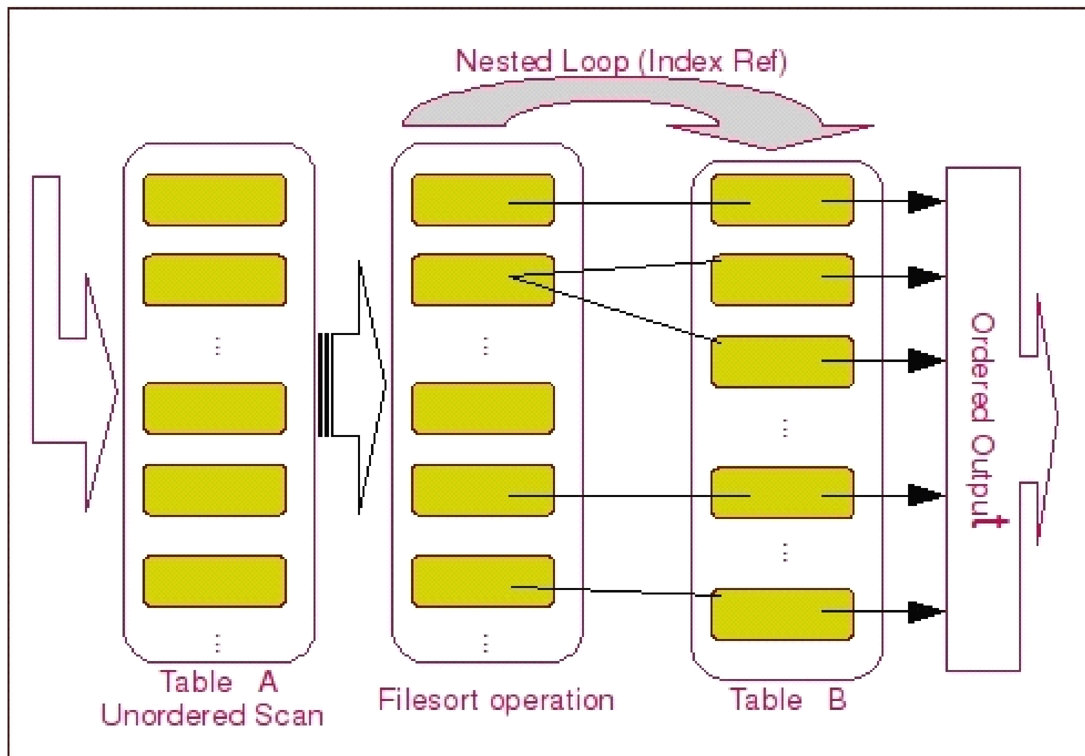
1. 取出满足过滤条件的用于排序条件的字段以及可以直接定位到行数据的行指针信息，在 Sort Buffer 中进行实际的排序操作，然后利用排好序之后的数据根据行指针信息返回表中取得客户端请求的其他字段的数据，再返回给客户端；
2. 根据过滤条件一次取出排序字段以及客户端请求的所有其他字段的数据，并将不需要排序的字段存放在一块内存区域中，然后在 Sort Buffer 中将排序字段和行指针信息进行排序，最后再利用排序后的行指针与存放在内存区域中和其他字段一起的行指针信息进行匹配合并结果集，再按照顺序返回给客户端。

上面第一种排序算法是 MySQL 一直以来就有的排序算法，而第二种则是从 MySQL4.1 版本才开始增加的改进版排序算法。第二种算法与第一种相比较，主要优势就是减少了数据的二次访问。在排序之后

不需要再一次回到表中取数据，节省了 IO 操作。当然，第二种算法会消耗更多的内存，正是一种典型的通过内存空间换取时间的优化方式。下面我们同样通过一个实例来看看当 MySQL 不得不使用排序算法的时候的执行计划，仅仅只是更改一下排序字段：

```
sky@localhost : example 10:09:06> explain
    -> select m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.subject\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: const
         rows: 4
        Extra: Using where; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: group_message_content_msg_id
          key: group_message_content_msg_id
      key_len: 4
          ref: example.m.id
         rows: 11
        Extra:
```

大概一看，好像整个执行计划并没有什么区别啊？但是细心的读者朋友可能已经发现，在 group_message 表的 Extra 信息中，多了一个“Using filesort”的信息，实际上这就是 MySQL Query Optimizer 在告诉我们，他需要进行排序操作才能按照客户端的要求返回有序的数据。执行图示如下：



这里我们看到了，MySQL 在取得第一个表的数据之后，先根据排序条件将数据进行了一次 filesort，也就是排序操作。然后再利用排序后的结果集作为驱动结果集来通过 Nested Loop Join 访问第二个表。当然，大家不要误解，这个 filesort 并不是说通过磁盘文件进行排序，仅仅只是告诉我们进行了一个排序操作。

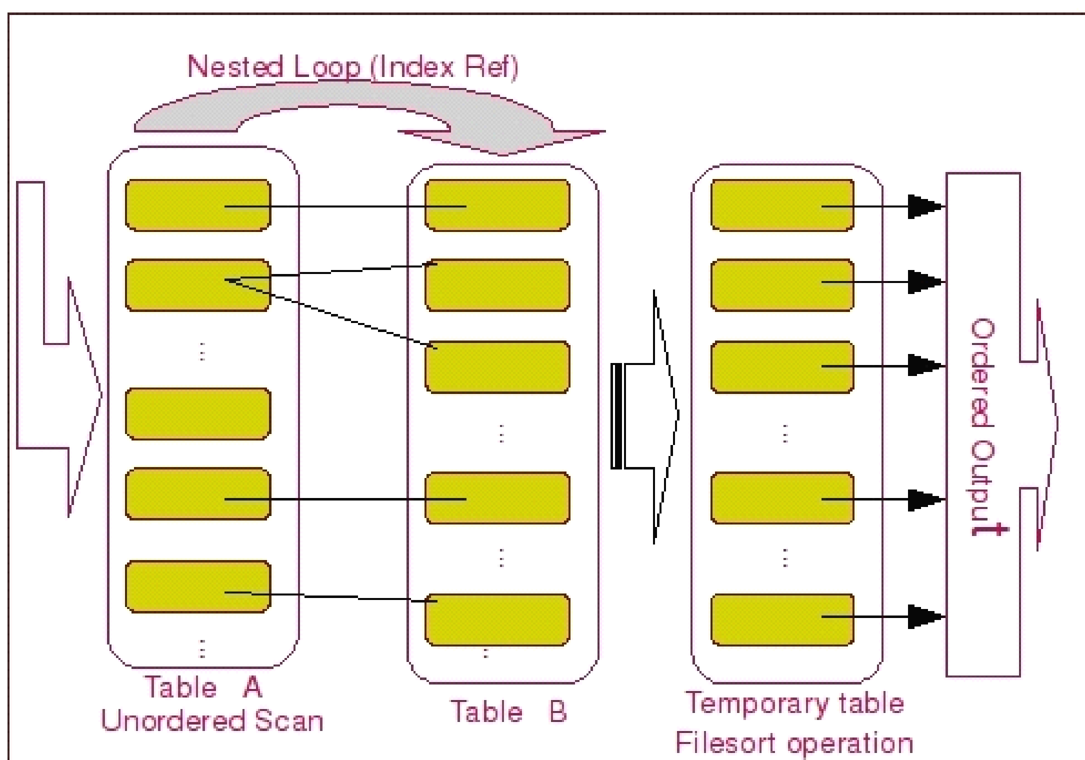
上面，我们看到了排序结果集来源仅仅只是单个表的比较简单的 filesort 操作。而在我们实际应用中，很多时候我们的业务要求可能并不是这样，可能需要排序的字段同时存在于两个表中，或者 MySQL 在经过一次 Join 之后才进行排序操作。这样的排序在 MySQL 中并不能简单的里利用 Sort Buffer 进行排序，而是必须先通过一个临时表将之前 Join 的结果集存放入临时表之后在将临时表的数据取到 Sort Buffer 中进行操作。下面我们通过再次更改排序要求来示例这样的执行计划，当我们选择通过 group_message_content 表上面的 content 字段来进行排序之后：

```
sky@localhost : example 10:22:42> explain
    -> select m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY c.content\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
```

序的结果集数据返回给客户端。

```
        key_len: 4
            ref: const
           rows: 4
          Extra: Using temporary; Using filesort
*************************** 2. row ***************************
             id: 1
    select_type: SIMPLE
          table: c
           type: ref
  possible_keys: group_message_content_msg_id
            key: group_message_content_msg_id
        key_len: 4
            ref: example.m.id
           rows: 11
          Extra:
```

这时候的执行计划中出现了“Using temporary”，正是因为我们的排序操作需要在两个表 Join 之后才能进行，下图展示了这个 Query 的执行过程：



首先是 Table A 和 Table B 进行 Join，然后结果集进入临时表，再进行 filesort，最后得到有序的结果集数据返回给客户端。

上面我们通过两个不同的示例展示了当 MySQL 无法避免要使用相应的排序算法进行排序操作的时候的实现原理。虽然在排序过程中所使用的排序算法有两种，但是两种排序的内部实现机制大体上差不

多。

当我们无法避免排序操作的时候，我们又该如何来优化呢？很显然，我们应该尽可能让 MySQL 选择使用第二种算法来进行排序。这样可以减少大量的随机 IO 操作，很大幅度的提高排序工作的效率。

1. 加大 max_length_for_sort_data 参数的设置；
   在 MySQL 中，决定使用第一种老式的排序算法还是新的改进算法的依据是通过参数 max_length_for_sort_data 来决定的。当我们所有返回字段的最大长度小于这个参数值的时候，MySQL 就会选择改进后的排序算法，反之，则选择老式的算法。所以，如果我们有充足的内存让 MySQL 存放需要返回的非排序字段的时候，可以加大这个参数的值来让 MySQL 选择使用改进版的排序算法。

2. 去掉不必要的返回字段；
   当我们的内存并不是很充裕的时候，我们不能简单的通过强行加大上面的参数来强迫 MySQL 去使用改进版的排序算法，因为如果那样可能会造成 MySQL 不得不将数据分成很多段然后进行排使用序，这样的结果可能会得不偿失。在这种情况下，我们就需要去掉不必要的返回字段，让我们的返回结果长度适应 max_length_for_sort_data 参数的限制。

3. 增大 sort_buffer_size 参数设置；
   增大 sort_buffer_size 并不是为了让 MySQL 可以选择改进版的排序算法，而是为了让 MySQL 可以尽量减少在排序过程中对需要排序的数据进行分段，因为这样会造成 MySQL 不得不使用临时表来进行交换排序。

GROUP BY 的实现与优化

由于 GROUP BY 实际上也同样需要进行排序操作，而且与 ORDER BY 相比，GROUP BY 主要只是多了排序之后的分组操作。当然，如果在分组的时候还使用了其他的一些聚合函数，那么还需要一些聚合函数的计算。所以，在 GROUP BY 的实现过程中，与 ORDER BY 一样也可以利用到索引。

在 MySQL 中，GROUP BY 的实现同样有多种（三种）方式，其中有两种方式会利用现有的索引信息来完成 GROUP BY，另外一种为完全无法使用索引的场景下使用。下面我们分别针对这三种实现方式做一个分析。

1. 使用松散（Loose）索引扫描实现 GROUP BY

何谓松散索引扫描实现 GROUP BY 呢？实际上就是当 MySQL 完全利用索引扫描来实现 GROUP BY 的时候，并不需要扫描所有满足条件的索引键即可完成操作得出结果。

下面我们通过一个示例来描述松散索引扫描实现 GROUP BY，在示例之前我们需要首先调整一下 group_message 表的索引，将 gmt_create 字段添加到 group_id 和 user_id 字段的索引中：

```
sky@localhost : example 08:49:45> create index idx_gid_uid_gc
    -> on group_message(group_id,user_id,gmt_create);
Query OK,  rows affected (0.03 sec)
Records: 96  Duplicates: 0  Warnings: 0
```

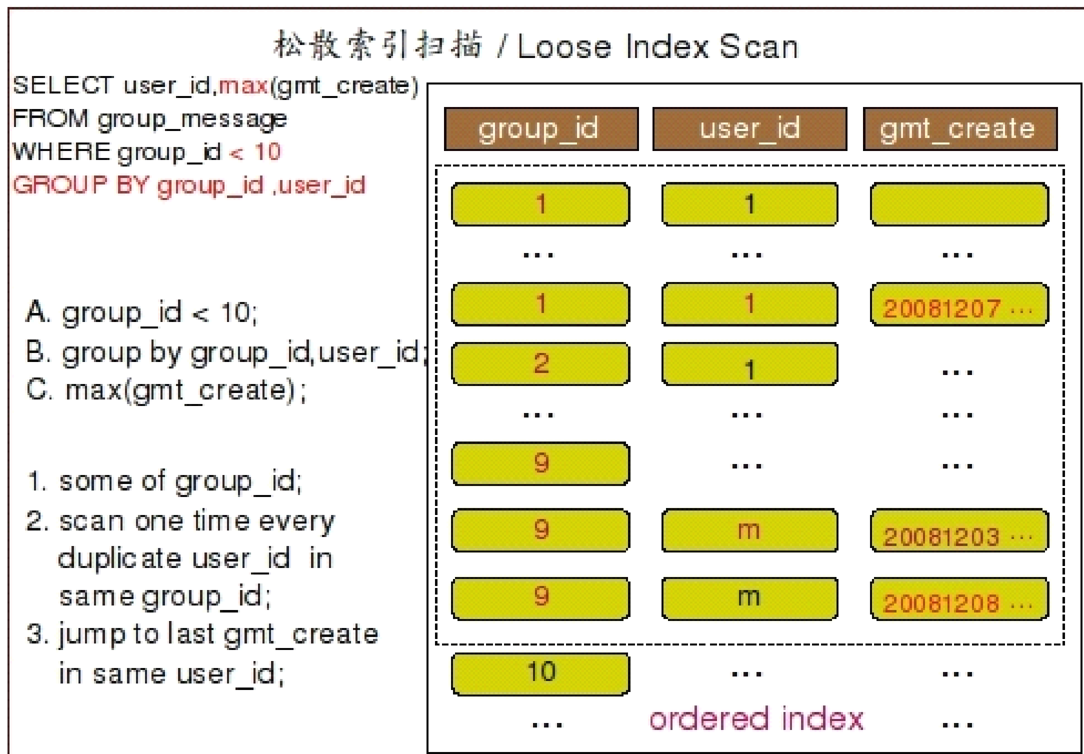
```
sky@localhost : example 09:07:30> drop index idx_group_message_gid_uid
    -> on group_message;
Query OK, 96 rows affected (0.02 sec)
Records: 96  Duplicates: 0  Warnings: 0
```

然后再看如下 Query 的执行计划：

```
sky@localhost : example 09:26:15> EXPLAIN
    -> SELECT user_id,max(gmt_create)
    -> FROM group_message
    -> WHERE group_id < 10
    -> GROUP BY group_id,user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 8
          ref: NULL
         rows: 4
        Extra: Using where; Using index for group-by
1 row in set (0.00 sec)
```

我们看到在执行计划的 Extra 信息中有信息显示“Using index for group-by”，实际上这就是告诉我们，MySQL Query Optimizer 通过使用松散索引扫描来实现了我们所需要的 GROUP BY 操作。

下面这张图片描绘了扫描过程的大概实现：



要利用到松散索引扫描实现 GROUP BY，需要至少满足以下几个条件：

- GROUP BY 条件字段必须在同一个索引中最前面的连续位置；
- 在使用 GROUP BY 的同时，只能使用 MAX 和 MIN 这两个聚合函数；
- 如果引用到了该索引中 GROUP BY 条件之外的字段条件的时候，必须以常量形式存在；

为什么松散索引扫描的效率会很高？

因为在没有 WHERE 子句，也就是必须经过全索引扫描的时候， 松散索引扫描需要读取的键值数量与分组的组数量一样多，也就是说比实际存在的键值数目要少很多。而在 WHERE 子句包含范围判断式或者等值表达式的时候， 松散索引扫描查找满足范围条件的每个组的第 1 个关键字，并且再次读取尽可能最少数量的关键字。

2. 使用紧凑（Tight）索引扫描实现 GROUP BY

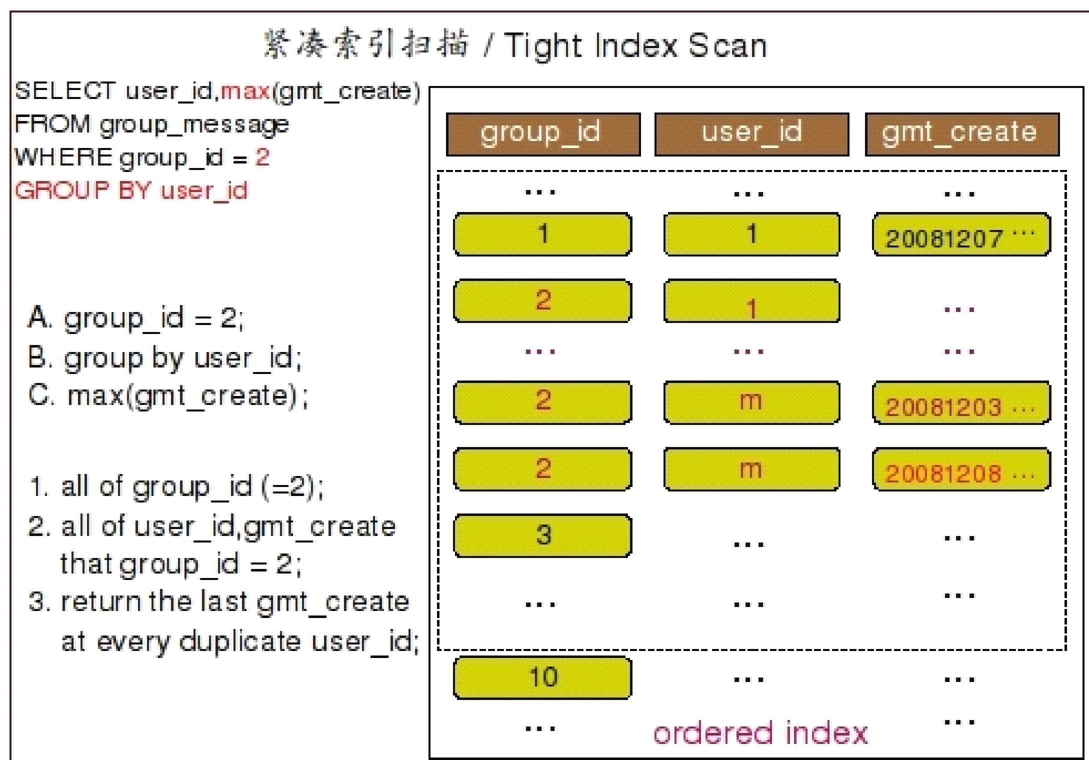
紧凑索引扫描实现 GROUP BY 和松散索引扫描的区别主要在于他需要在扫描索引的时候，读取所有满足条件的索引键，然后再根据读取恶的数据来完成 GROUP BY 操作得到相应结果。

```
sky@localhost : example 08:55:14> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id = 2
    -> GROUP BY user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
```

```
          type: ref
possible_keys: idx_group_message_gid_uid,idx_gid_uid_gc
           key: idx_gid_uid_gc
       key_len: 4
           ref: const
          rows: 4
         Extra: Using where; Using index
1 row in set (0.01 sec)
```

这时候的执行计划的 Extra 信息中已经没有“Using index for group-by”了，但并不是说 MySQL 的 GROUP BY 操作并不是通过索引完成的，只不过是需要访问 WHERE 条件所限定的所有索引键信息之后才能得出结果。这就是通过紧凑索引扫描来实现 GROUP BY 的执行计划输出信息。

下面这张图片展示了大概的整个执行过程：



在 MySQL 中，MySQL Query Optimizer 首先会选择尝试通过松散索引扫描来实现 GROUP BY 操作，当发现某些情况无法满足松散索引扫描实现 GROUP BY 的要求之后，才会尝试通过紧凑索引扫描来实现。

当 GROUP BY 条件字段并不连续或者不是索引前缀部分的时候，MySQL Query Optimizer 无法使用松散索引扫描，设置无法直接通过索引完成 GROUP BY 操作，因为缺失的索引键信息无法得到。但是，如果 Query 语句中存在一个常量值来引用缺失的索引键，则可以使用紧凑索引扫描完成 GROUP BY 操作，因为常量填充了搜索关键字中的“差距”，可以形成完整的索引前缀。这些索引前缀可以用于索引查找。而如果需要排序 GROUP BY 结果，并且能够形成索引前缀的搜索关键字，MySQL 还可以避免额外的排序操作，因为使用有顺序的索引的前缀进行搜索已经按顺序检索到了所有关键字。

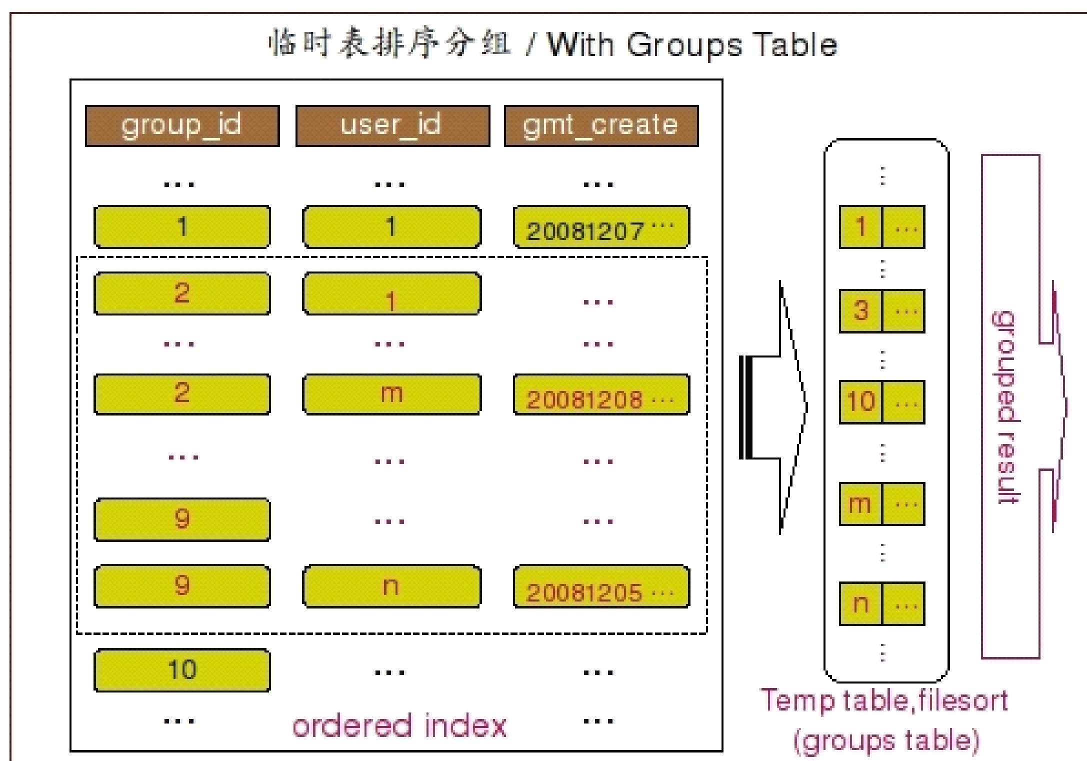
3. 使用临时表实现 GROUP BY

MySQL 在进行 GROUP BY 操作的时候要想利用所有，必须满足 GROUP BY 的字段必须同时存放于同一个索引中，且该索引是一个有序索引（如 Hash 索引就不能满足要求）。而且，并不只是如此，是否能够利用索引来实现 GROUP BY 还与使用的聚合函数也有关系。

前面两种 GROUP BY 的实现方式都是在有可以利用的索引的时候使用的，当 MySQL Query Optimizer 无法找到合适的索引可以利用的时候，就不得不先读取需要的数据，然后通过临时表来完成 GROUP BY 操作。

```
sky@localhost : example 09:02:40> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id > 1 and group_id < 10
    ->  GROUP BY user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: idx_group_message_gid_uid,idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 32
        Extra: Using where; Using index; Using temporary; Using filesort
```

这次的执行计划非常明显的告诉我们 MySQL 通过索引找到了我们需要的数据，然后创建了临时表，又进行了排序操作，才得到我们需要的 GROUP BY 结果。整个执行过程大概如下图所展示：



当 MySQL Query Optimizer 发现仅仅通过索引扫描并不能直接得到 GROUP BY 的结果之后，他就不得不选择通过使用临时表然后再排序的方式来实现 GROUP BY 了。

在这样示例中即是这样的情况。 group_id 并不是一个常量条件，而是一个范围，而且 GROUP BY 字段为 user_id。所以 MySQL 无法根据索引的顺序来帮助 GROUP BY 的实现，只能先通过索引范围扫描得到需要的数据，然后将数据存入临时表，然后再进行排序和分组操作来完成 GROUP BY。

对于上面三种 MySQL 处理 GROUP BY 的方式，我们可以针对性的得出如下两种优化思路：

1. 尽可能让 MySQL 可以利用索引来完成 GROUP BY 操作，当然最好是松散索引扫描的方式最佳。在系统允许的情况下，我们可以通过调整索引或者调整 Query 这两种方式来达到目的；
2. 当无法使用索引完成 GROUP BY 的时候，由于要使用到临时表且需要 filesort，所以我们必须要有足够的 sort_buffer_size 来供 MySQL 排序的时候使用，而且尽量不要进行大结果集的 GROUP BY 操作，因为如果超出系统设置的临时表大小的时候会出现将临时表数据 copy 到磁盘上面再进行操作，这时候的排序分组操作性能将是成数量级的下降；

至于如何利用好这两种思路，还需要大家在自己的实际应用场景中不断的尝试并测试效果，最终才能得到较佳的方案。此外，在优化 GROUP BY 的时候还有一个小技巧可以让我们在有些无法利用到索引的情况下避免 filesort 操作，也就是在整个语句最后添加一个以 null 排序（ORDER BY null）的子句，大家可以尝试一下试试看会有什么效果。

DISTINCT 的实现与优化

DISTINCT 实际上和 GROUP BY 的操作非常相似，只不过是在 GROUP BY 之后的每组中只取出一条记

录而已。所以，DISTINCT 的实现和 GROUP BY 的实现也基本差不多，没有太大的区别。同样可以通过松散索引扫描或者是紧凑索引扫描来实现，当然，在无法仅仅使用索引即能完成 DISTINCT 的时候，MySQL 只能通过临时表来完成。但是，和 GROUP BY 有一点差别的是，DISTINCT 并不需要进行排序。也就是说，在仅仅只是 DISTINCT 操作的 Query 如果无法仅仅利用索引完成操作的时候，MySQL 会利用临时表来做一次数据的“缓存”，但是不会对临时表中的数据进行 filesort 操作。当然，如果我们在进行 DISTINCT 的时候还使用了 GROUP BY 并进行了分组，并使用了类似于 MAX 之类的聚合函数操作，就无法避免 filesort 了。

下面我们就通过几个简单的 Query 示例来展示一下 DISTINCT 的实现。

1. 首先看看通过松散索引扫描完成 DISTINCT 的操作：

```
sky@localhost : example 11:03:41> EXPLAIN SELECT DISTINCT group_id
    -> FROM group_message\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: range
possible_keys: NULL
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 10
        Extra: Using index for group-by
1 row in set (0.00 sec)
```

我们可以很清晰的看到，执行计划中的 Extra 信息为“Using index for group-by”，这代表什么意思？为什么我没有进行 GROUP BY 操作的时候，执行计划中会告诉我这里通过索引进行了 GROUP BY 呢？其实这就是于 DISTINCT 的实现原理相关的，在实现 DISTINCT 的过程中，同样也是需要分组的，然后再从每组数据中取出一条返回给客户端。而这里的 Extra 信息就告诉我们，MySQL 利用松散索引扫描就完成了整个操作。当然，如果 MySQL Query Optimizer 要是能够做的再人性化一点将这里的信息换成“Using index for distinct”那就更好更容易让人理解了，呵呵。

2. 我们再来看看通过紧凑索引扫描的示例：

```
sky@localhost : example 11:03:53>  EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id = 2\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
```

```
        key_len: 4
            ref: const
           rows: 4
          Extra: Using WHERE; Using index
1 row in set (0.00 sec)
```

这里的显示和通过紧凑索引扫描实现 GROUP BY 也完全一样。实际上，这个 Query 的实现过程中，MySQL 会让存储引擎扫描 group_id = 2 的所有索引键，得出所有的 user_id，然后利用索引的已排序特性，每更换一个 user_id 的索引键值的时候保留一条信息，即可在扫描完所有 gruop_id = 2 的索引键的时候完成整个 DISTINCT 操作。

3. 下面我们在看看无法单独使用索引即可完成 DISTINCT 的时候会是怎样：

```
    sky@localhost : example 11:04:40> EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10\G
*************************** 1. row ***************************
             id: 1
    SELECT_type: SIMPLE
          table: group_message
           type: range
  possible_keys: idx_gid_uid_gc
            key: idx_gid_uid_gc
        key_len: 4
            ref: NULL
           rows: 32
          Extra: Using WHERE; Using index; Using temporary
1 row in set (0.00 sec)
```

当 MySQL 无法仅仅依赖索引即可完成 DISTINCT 操作的时候，就不得不使用临时表来进行相应的操作了。但是我们可以看到，在 MySQL 利用临时表来完成 DISTINCT 的时候，和处理 GROUP BY 有一点区别，就是少了 filesort。实际上，在 MySQL 的分组算法中，并不一定非要排序才能完成分组操作的，这一点在上面的 GROUP BY 优化小技巧中我已经提到过了。实际上这里 MySQL 正是在没有排序的情况下实现分组最后完成 DISTINCT 操作的，所以少了 filesort 这个排序操作。

4. 最后再和 GROUP BY 结合试试看：

```
sky@localhost : example 11:05:06> EXPLAIN SELECT DISTINCT max(user_id)
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10
    -> GROUP BY group_id\G
*************************** 1. row ***************************
             id: 1
    SELECT_type: SIMPLE
```

```
        table: group_message
         type: range
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 32
        Extra: Using WHERE; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

最后我们再看一下这个和 GROUP BY 一起使用带有聚合函数的示例，和上面第三个示例相比，可以看到已经多了 filesort 排序操作了，因为我们使用了 MAX 函数的缘故。

对于 DISTINCT 的优化，和 GROUP BY 基本上一致的思路，关键在于利用好索引，在无法利用索引的时候，确保尽量不要在大结果集上面进行 DISTINCT 操作，磁盘上面的IO操作和内存中的IO操作性能完全不是一个数量级的差距。

8.7 小结

本章重点介绍了 MySQL Query 语句相关的性能调优的部分思路和方法，也列举了部分的示例，希望能够帮助读者朋友在实际工作中开阔一点点思路。虽然本章涉及到的内容包含了最初的索引设计，到编写高效 Query 语句的一些原则，以及最后对语句的调试，但 Query 语句的调优远不只这些内容。很多的调优技巧，只有到在实际的调优经验中才会真正体会，真正把握其精髓。所以，希望各位读者朋友能多做实验，以理论为基础，以事实为依据，只有这样，才能不断提升自己对 Query 调优的深入认识。

第 9 章 MySQL 数据库 Schema 设计的性能优化

前言：

很多人都认为性能是在通过编写代码（程序代码或者是数据库代码）的过程中优化出来的，其实这是一个非常大的误区。真正影响性能最大的部分是在设计中就已经产生了的，后期的优化很多时候所能够带来的改善都只是在解决前妻设计所遗留下来的一些问题而已，而且能够解决的问题通常也比较有限。本章将就如何在 MySQL 数据库 Schema 设计的时候保证尽可能的高效，尽可能减少后期的烦恼。

9.1 高效的模型设计

最规范的就一定是最合理的吗？

在数据库 Schema 设计理论方面，一直有一个被大家奉为“葵花宝典”的规范化范式理论。通过范式理论所设计的数据库 Schema 逻辑清晰，关系明确，扩展方便，就连存储的数据量也做到了尽可能的少，尤其是当范式级别较高的时候，几乎找不到任何的冗余数据。在很多人眼里，数据库 Schema 满足的范式级别越高则该 Schema 设计的越优秀。

但是，很多人忽略了一点，那就是产生该理论的时期和出发点。关系性数据库的规范化范式理论诞生于上世纪七十年代初，最根本的目的是让数据库中尽量的去除数据的冗余，保持数据的一致，使数据的修改简单。

实际上，尽量去除数据的冗余不仅仅是为了让我们查询相同的数据量的时候能够多返回几条记录，还有一个很重要的原因就是在当时的那个年代，数据的存储空间是及其昂贵的，而且存储设备的容量也都非常的小，这一点在硬件存储设备发展如此迅速的如今，空间大小已经不再是太大的问题了。

而范式理论中的数据一致性和使数据修改简单保证主要是依靠添在数据库中添加各种约束来保证，而各种约束对于数据库来说本身其实就是一个非常消耗资源的事情。

所以，对于基于性能的数据库 Schema 设计，我们并不能完全以规范化范式理论来作为唯一的指导。在设计过程中，应该从实际需求出发，以性能提升为根本目标来展开设计工作，很多时候为了尽可能提高性能，我们必须做反范式设计。

适度冗余 - 让 Query 尽两减少 Join

熟悉 MySQL 的优化器的读者可能清楚，MySQL 的优化器虽然号称使用了新一代的优化器技术实现的非常优秀，但是由于目前 MySQL 所收集的数据统计信息还不是特别的多，所以起表现并不是特别的让人满意，也并非如 MySQL 官方所宣传的那样智能。虽然处理普通 Join 的时候一般都能比较智能的得到比较高效的执行计划，但是当遇到一些自查询或者较为复杂的 Join 的时候，很容易出现不太合理的执行计划，不少时候对各表的访问顺序选择的并不合适，造成复杂 Query 的整体执行效率低下。

所以，为了让我们的 Query 执行计划尽可能的最优化，最直接有效的方式就是尽量减少 Join，而要

减少 Join，我们就不可避免的需要通过表字段的冗余来实现。

这里我们继续通过“影响 MySQL Server 性能的相关因素” 一章中 “Schema 设计对性能的影响”这一节的一个例子来进一步分析资源消耗的差异。方案一中的 group_message 表中仅保存了发布信息者的 ID 信息，而通过冗余优化之后的 group_message 表中增加了发布信息者的 nick_name 信息存为 author。

优化前实现列表功能的 Query 和执行计划（group_message_bad 是优化前的表，优化后为 group_message 表）：

```
sky@localhost : example 09:13:41> explain
    -> SELECT t.id, t.subject,user.id, user.nick_name
    ->     FROM (
    ->         SELECT id, user_id, subject
    ->         FROM group_message
    ->         WHERE group_id = 1
    ->         ORDER BY gmt_modified DESC LIMIT 1,10
    ->     ) t, user
    ->     WHERE t.user_id = user.id\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: user
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
```

```
        table: group_message
         type: ALL
possible_keys: group_message_gid_ind
          key: group_message_gid_ind
      key_len: 4
          ref:
         rows: 1
        Extra: Using filesort
```

优化后实现列表功能的 Query 和执行计划：

```
sky@localhost : example 09:14:06> explain
    -> SELECT t.id, t.subject, t.user_id, t.author
    ->     FROM group_message t
    ->     WHERE group_id = 1
    ->     ORDER BY gmt_modified DESC LIMIT 1,10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ref
possible_keys: group_message_gid_ind
          key: group_message_gid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra: Using where; Using filesort
```

从优化前和优化后的执行计划可以看出两者的差别非常大的，优化前必须检索 2 个表（group_message 和 user）才能得到结果，而优化后只需要检索 group_message 一个表就可以完成，因为我们将“作者”信息冗余到了 group_message。

从数据库范式理论来看，这样的设计是不合理的。因为可能造成 user 表和 group_message 表中的用户昵称数据不一致。每次更新用户昵称的时候，都需要更新两个表的数据，为了尽可能让两者数据保证一致，应用程序中需要处理更多的逻辑。但是，从性能角度来看的话，这种冗余是非常有价值的，虽然我们的数据更新逻辑复杂了，但是我们在考虑更新带来的附加成本的时候，还应该考虑我们到底会有多少更新发生在用户昵称上面呢？我们需要考虑的是一个系统的整体性能，而不是系统中单个行为的性能。就像示例中的昵称数据，虽然更新的成本增加了，但是查询的效率提高了，而且发生示例中查询的频率要远大于更新的频率，通过少部分操作的成本投入换取更大的性能收获，实际上是我们系统性能优化中经常使用的策略。

在大部分应用系统中，类似于上面示例中的这种查询频繁但是更新较少的数据非常非常多，很多时候如果我们一味的追求范式化理论的 Schema 设计在高性能要求的系统中是非常不合适的。我个人认为，数据库的规范化理论其实质是在概念上的单一化，虽然规范后的数据库中的表一般都较小，使表中相关列最少。这虽然可能在某些情况下增强了数据库的可维护性，但在系统要完成一些数据的查询检索时，

可能要用复杂的 Join 才能实现，这势必会造成查询检索的性能低下。如果我们通过拆分 Join，通过多次简单的查询来在应用中实现 Join 逻辑，那所带来的网络开销将会是非常巨大的。

大字段垂直分拆 – summary 表优化

实际上，在上面的示例中我们同时还用到了另外一种优化策略，也就是“大字段垂直拆分”策略。大字段垂直拆分策略相对于前面介绍的适度冗余策略在做法上可以说产不多是完全相反的做法。适度冗余策略是将别的表中的字段拿过来在自己身上也存一份数据，而大字段垂直拆分简单来说就是将自己身上的字段拆分出去放在另外（单独）的表里面。

可能很多读者朋友都会有疑惑了，我们刚刚才分析出了将别的字段拿过来放自己表里面为什么现在又要将自己的字段分出去呢？这样不是有些自相矛盾了吗？

其实并没有任何矛盾，前面我们将别人的字段那过来，是因为我们很多时候的查询需要使用该字段，为了减少 Join 带来的性能消耗才拿过来的。而我们将大字段拿出去，也是将一些我们在大部分查询中并不需要使用该字段的时候才会拿出去。而且，在我们拿出去之前，我们肯定会通过全面的评估比较之后才能做出拆分出去的决定。

那到底什么样的字段适合于从表中拆分出去呢？

首要肯定是大字段。为什么？原因很简单，就是因为他的大。大字段一般都是存放着一些较长的 Detail 信息，如文章的内容，帖子的内容，产品的介绍等等。

其次是和表中其他字段相比访问频率明显要少很多。由于大字段存放的内容较多，大部分情况都是占整条记录的 80%以上，而数据库中数据在数据文件中的格式一般都是以一条一条记录为单位来存放。也就是说，如果我们要查询某些记录的某几个字段，数据库并不是只需要访问我们需要查询的哪几个字段，而是需要读取其他所有字段（可以在索引中完成整个查询的情况除外），也无法做到只读取我们需要的几个字段的数据。这样，我们就不得不读取包括大字段在内的很多并不相干的数据。而由于大字段所占的空间比例非常大，自然所浪费的 IO 资源也就非常之大了。

在这样的场景下，我们就需要将该大字段从原表中拆分出来，通过单独的表进行存放，让我们在访问其他数据的时候大大降低 IO 访问，从而使性能得到较大的改善。

可能有人会疑惑，虽然移出之后访问其他字段的效率提高了，但是当我们需要大字段的信息的时候，我们就无法避免的需要通过 Join 来实现，而使用 Join 之后的处理效率可能会大打折扣的。其实这个担心是很合理的，这也就是我们在分拆出大字段之前需要还需要考虑的第二个因素，访问频率的因素了。前面我们就介绍了，决定是否要分拆出，出了“大”之外，还要“频率低”才行，当然，这里的“频率低”只是“相对频率”而已。而且，这种分拆之后的两个表的关系都是完全确定的一一对应关系，使用 Join 在性能方面的影响也并不是特别的大。

那我们在移出大字段的同时，是否还需要将其他字段也一并移出呢？其实如果我们已经确定有大字段需要分拆出主表的时候，对于其他的字段，只要满足访问频率和大字段一样相对于表中其他字段要低很多的都可以和大字段同时分拆出来。

实际上，在有些时候，我们甚至都不一定非要大字段才能进行垂直分拆。在有些场景下，有的表中大部分字段平时都很少访问，而其中的某几个字段却是访问频率非常高。对于这种表，也非常适合通过垂直分拆来达到优化性能的目的。

在“Schema 设计对性能的影响”一节中的示例中，实际上是有两处用到了“垂直分拆”这个优化策略。一处是 group_message_bad 表中的 content 大字段从原表中分拆出来为 group_message_content 表。另一处就是将原 user_bad 表中虽然不大但是平时使用很少的字段拆分出来新增了 user_profile 表。

大表水平分拆 - 基于类型的分拆优化

“大表水平拆分”策略在性能优化方面可能被人使用的频率并不是太多，但是如果使用得当，很可能会给我们带来不小的惊喜。

我们还是直接通过实例来说明问题吧。假设我们将前面示例中的需求稍微做一下扩展，我们希望 group 系统总管理员能够发布系统消息，而且在每一个 group 的讨论帖的没一页都能置顶显示。

在得到该需求之后，我们的第一反应肯定是通过在 group_message 表中增加一个标识列，用来存放帖子的类型，标识出是普通会员的讨论贴还是系统管理员的置顶帖。然后在每个列表展示页面都通过对 group_message 表的两次查询（一次置顶信息，一次普通讨论帖）然后在应用程序中合并再展示。这样的结果是由于整个 group_message 表的数据较大，查询置顶信息的 Query 成本会相对有些高。

下面我们换一个思路来考虑一下这个问题：
首先，置顶信息和其他讨论帖完全不会产生任何关联交互；
其次，置顶信息的变化相对于其他讨论帖来说变化很少；
再次，置顶信息的访问频率非常高；
最后，置顶信息的量和普通讨论帖来比非常之少；

通过上面的这几个分析，如果我们将置顶信息单独存放在普通讨论帖之外的其他表里面，首先不会带来什么附加的性能消耗，而且可以使每次检索置顶信息的成本都有所下降。由于访问频率非常的高，则因为每次检索置顶信息的成本下降而得到较大的节省。数量少而且变化不怎么频繁的特点则非常适合使用 MySQL 的 Query Cache，而如果和普通讨论帖在一起由于普通讨论帖的频繁变化带来 group_message 表相关的 Query Cache 失效问题会让他无法使用 Query Cache 功能。

通过上面的分析，我们很容易得出一个更为优化的方案来存放这些置顶信息，那就是新增一张类似于 group_message 的表来专门存放置顶信息，我们暂且命名为 top_message 如下：

```
sky@localhost : example 10:49:20> desc top_message;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| id           | int(11)      | NO   |     | 0       |       |
| gmt_create   | datetime     | NO   |     | NULL    |       |
| gmt_modified | datetime     | NO   |     | NULL    |       |
| user_id      | int(11)      | NO   |     | NULL    |       |
| author       | varchar(32)  | NO   |     | NULL    |       |
```

```
| subject       | varchar(128) | NO    |     | NULL     |       |
+--------------+-------------+------+-----+---------+-------+
```

由于是全局的，所以省略了 group_id 信息，而 content 信息，还是同样可以存放在 group_message_content 表中。

上面仅仅只是一个示例，可能在实际应用中并不是如此的简单，但这里只是给大家一个思路，让大家知道如何通过大表的水平拆分来对通过优化 Schema 设计提供系统的整体性能。在很多大型的应用中，由于数据量非常庞大，并发访问又非常高，到达单台主机都无法支撑单个表的访问的时候，常常会通过这种大表的水平拆分，存放在多台主机的多个数据库中实现整体扩展性的提升，这方面的内容我们将在“架构设计”部分的“可扩展设计之数据切分”章节再做更为详细的介绍。

统计表 – 准实时优化

统计表的准实时优化策略实际上我们在“影响 MySQL Server 性能的相关因素”一章的“商业需求对性能的影响”部分有提出过。简单来说就是通过定时统计数据来替代实时统计查询。

为什么要准实时？
很多人看到这个优化策略之后可能都会提出这样的质疑，为什么要改变需求将“可以实时”的统计信息做成准实时的呢？原因很简单，因为实时统计的性能消耗成本太高。因为每一次展示（也就是每一次刷新页面）都需要进行统计计算，带来大量的重复资源浪费。而做成准实时的统计信息之后，我们每次只需要访问很小的数据量即可，不需要频繁的统计计算的工作。

当然，并不是所有的统计数据都适合于通过准实时的统计表优化策略来实现的，即使我们希望，产品经理们也不会允许，即使产品经理们也希望那样，我们的使用者肯定也会不同意。

什么类型的统计信息适合通过准实时统计表来优化实现？
首先，统计信息的准确性要求并不是特别的严格；
其次，统计信息对时间并不是太敏感；
再次，统计信息的访问非常频繁，重复执行较多；
最后，参与统计数据量较大；

看看上面的要求，还真不少。不过，大家所维护的系统中确实很可能存在这样的统计数据展示功能。如系统当前在线人数，论坛系统当前总帖数、回帖数等，多条件大结果集查询页面的总结果数以及总页数，某些虚拟积分的 top n 排名等等。

这些统计的计算都会设计到大量的数据，同时也需要大量的计算资源，访问频率也都非常的高。如果都通过实时统计，恐怕只要数据量稍微大一些，都会带来非常大的硬件资源开销。但在短时间内的不够精确，又并不会带来太大用户体验的降低。所以完全可以通过定时任务程序，没隔一定时间段进行一次统计后存放在专门设计的统计表中。这样，在统计数据需要展示的时候，我们只需要从统计好的结果数据中取出即可。这样每次统计数据的展示性能将会成数量级的提升，反而会使整体的用户体验上升。

9.2 合适的数据类型

实际上在很多数据库的设计优化文档中都有关于通过优化数据类型的优化说明内容，在 MySQL 中，我们同样也可以通过数据类型的优化达到优化整个 Schema 设计的目的。

优化数据类型提高性能的主要原理在于以下几个方面：

1. 通过选用更“小”的数据类型减少存储空间，使查询相同数据需要的 IO 资源降低；
2. 通过合适的数据类型加速数据的比较；

下面我们还是通过分析一些常用数据类型的数据存储格式和长度来看看哪些数据类型可以在优化中利用上吧。

数字日期类型

我们先来看看存放长度基本固定的一些数据类型的存储长度和取值范围。

类型（同义词）	存储长度	最小值（无符号）	最大值（无符号）
整型数字			
TINYINT	1	－128（0）	127（255）
SMALLINT	2	-32768（0）	32767（65535）
MEDIUMINT	3	-8388608（0）	8388607（16777215）
INT（INTEGER）	4	-2147483648（0）	2147483647（4294967295）
BIGINT	8	-9223372036854775808（0）	9223372036854775807（18446744073709551615）
小数支持			
FLOAT[(M[,D])]	4 or 8	-3.402823466E+38-1.175494351E-38 0 1.175494351E-38～3.402823466E+38	
DOUBLE[(M[,D])]（REAL, DOUBLE PRECISION）	8	-1.7976931348623157E+308～-2.2250738585072014E-308; 0 2.2250738585072014E-308～ 1.7976931348623157E+308	
时间类型			
DATETIME	8	1001-01-01 00:00:00	9999-12-31 23:59:59
DATE	3	1001-01-01	9999-12-31
TIME	3	00:00:00	23:59:59
YEAR	1	1001	9999
TIMESTAMP	4	1970-01-01 00:00:00	

对于数字类型，这里分别列出了整数类型和小数类型，也就是浮点数类型。实际上，还有一类通过二进制格式以字符串来存放的数字类型如 DECIMAL(DEC)[(M[,D])]，NUMERIC[(M[,D])]，由于其存放长度主要通过其定义时候的的 M 所决定，M 定义为多大，则实际存放就有多长。M 代表整个位数长度，而 D 则表示小数点后的位数，默认 M 为 10，D 为 0。一般来说，主要用在固定精度的场合，由于其存放长度较大，而且考虑到这种数据完全可以变化形式以整数存放，所以笔者个人并不是特别推荐。

对于数字的存储，一般使用到浮点型数据的场合也不应该太多。主要出于两个原因，一个是浮点型数据本身实际上是一个并不精确的数字，只是一个近似值，另一个原因就是完全可以通过乘以一个固定的系数转换为整型数据来存放。这样不仅可以解决数据不精确的问题，同时也让数据的处理更为高效。

时间存储格式总类并不是太多，我们常用的主要就是DATETIME，DATE和TIMESTAMP这三种了。从存储空间来看TIMESTAMP最少，四个字节，而其他两种数据类型都是八个字节，多了一倍。而TIMESTAMP的缺点在于他只能存储从1970年之后的时间，而另外两种时间类型可以存放最早从1001年开始的时间。如果有需要存放早于1970年之前的时间的需求，我们必须放弃TIMESTAMP类型，但是只要我们不需要使用1970年之前的时间，最好尽量使用TIMESTAMP来减少存储空间的占用。

上面所列出的主要是一些存放固定长度，且我们平时可能常用到的一些类型。通过这个对照表格，我们可以很直观的看出哪种类型占用的存储空间大，哪种占用的空间小。这样，在数据类型选择的时候，我们就可以结合各种类型的存储范围以及业务中可能存在的数据作出对应，然后选择存储空间最先的类型来使用。

字符存储类型

我们再来看看存放字符的数据类型。

类型	存储占用最大空间
CHAR[(M)]	255 characters(independent of charset)
VARCHAR[(M)]	65535 bytes or 255 characters
TINYTEXT[(M)]	255 characters (sigle-byte)
TEXT[(M)]	65535 characters (sigle-byte)
MEDIUMTEXT[(M)]	16777215 characters (sigle-byte)
LONGTEXT[(M)]	4294967295 characters (sigle-byte)

CHAR[(M)]类型属于静态长度类型，存放长度完全以字符数来计算，所以最终的存储长度是基于字符集的，如latin1则最大存储长度为255字节，但是如果使用gbk则最大存储长度为510字节。CHAR类型的存储特点是不管我们实际存放多长数据，在数据库中都会存放M个字符，不够的通过空格补上，M默认为1。虽然CHAR会通过空格补齐存放的空间，但是在访问数据的时候，MySQL会忽略最后的所有空格，所以如果我们的实际数据中如果在最后确实需要空格，则不能使用CHAR类型来存放。在MySQL5.0.3之前的版本中，如果我们定义CHAR的时候M值超过255，MySQL会自动将CHAR类型进行转换为可以存入对应数据量的TEXT类型，如CHAR(1000)会自动转换为TEXT，CHAR(10000)则会转为MEDIUMTEXT。而从MySQL5.0.3开始，所有超过255的定义MySQL都会直接拒绝并给出错误信息，不再自动转换。

VARCHAR[(M)]属于动态存储长度类型，仅存占用实际存储数据的长度。其存放的最大长度与MySQL版本有关，在5.0.3之前的版本VARCHAR以字符数控制最存储的最大长度，最大只能存放255个字符，占用存储空间的实际大小与字符集有关。但是从5.0.3开始，VARCHAR的最大存储限制已经更改为字节数限制了，扩展到可以存放65535 bytes的数据，不同的字符集可能存放的字符数并不一样。也就是说，在MySQL5.0.3之前的版本，M所代表的是字符数，而从5.0.3版本开始，M的代表意思已经是字节数了。VARCHAR的存储特点是不管我们设定M为多大的值，真正占用的存储空间都只有我们所存入的实际数据的大小，和CHAR不同的是VARCHAR会保留我们存入数据最后的空格，也就是说我们存入是什么样，MySQL返回给我们的也会是什么样。在VARCHAR类型字段的数据中，MySQL会在每个VARCHAR数据中使用1个或者2个字节用来存放VARCHAR数据的实际长度，当我们的实际数据在255字节之内的时候，会使用1字节来存放实际长度，而大于255字节的时候，则需要使用2字节来存放。

TINYTEXT，TEXT，MEDIUMTEXT和LONGTEXT这四种类型同属于一种存储方式，都是动态存储长度类

型，不同的仅仅是最大长度的限制。四种类型的定义都是通过最大字符数来限制，但是他们的字符数限制实际上是可以理解为字节数限制的，因为当我们使用多字节字符集的时候，实际能存放的字符书并没最大字符数那么多，而是以单字节字符来计算的字符数。此外，由于是动态存储长度类型，所以和VARCHAR一样，每个字段数据之前都需要一个存放实际长度的空间。TINYTEXT需要1个字节来存放，TEXT需要2个字节，MEDIUMTEXT和LONGTEXT则分别需要3个和4个字节来存放实际数据长度。实际上，出了MySQL内嵌的最大长度限制之外，他们还受到客户端与服务器端的网络通信缓冲区最大值（max_allowed_packet）的限制。

这四种TEXT类型和CHAR及VARCHAR在实际使用中存在几个不一样的地方：

- 不能设置默认值；
- 只有TEXT可以使用TEXT[(M)]这样的方式通过M设置大小；
- 基于这四种类型的索引必须指定前缀长度；

其他常用类型

除了上面这些字段类型之外会被我们经常使用到之外，我们还会使用到的数据类型主要有以下这些。

类型	存储占用最大空间
BIT[(M)]	(M+7)/8 bytes ,最大(64+7)/8
SET('v1','v2'...)	1,2,4 or 8 bytes（取决于存储值的数目，最大64个值）
ENUM('v1','v2'...)	1 or 2 bytes （取决于存储值的数目，最大65535个值）

对于BIT类型，M表示每个值的bits数目，默认为1，最大为64 bits。对于MySQL来说这是一个新的类型，因为从MySQL5.0.3才开始真正实现（在之前实际上是TINYINT（1）），而且仅仅支持MyISAM存储引擎，但是从MySQL5.0.5开始Memory，Innodb和NDB Cluster存储引擎也开始“支持”了。在MyISAM中，BIT的存储空间很小，是真正的实现了通过bit来存储，但是在其他的一些存储引擎中就不一样了，因为他们是转换为最小的INT类型存储的，所以占用的空间也没有节省，还不如直接使用INT类的数据类型存放来得直观。

对于SET和ENUM类型，主要内容基本处于较少变化状态且值比较少的字段。虽然这两个字段所占用的存储空间都较少，但是由于在使用方面较其他的数据类型要略为复杂一些，所以在实际环境中一般使用还是较少。

谁都知道，数据量（这里主要指数据记录条数）的增加肯定会让数据库的检索查询效率降低。所以很多时候人们大都希望通过减少数据库中关键表的记录条数来获得数据库性能的提升。实际上，除了这种通过控制数据记录条数来控制数据总量的办法之外，我们还可以通过选择更小的数据类型来让数据库通过更小的空间存放相同的数据量，这对于检索同样的数据所带来的IO消耗自然会降低，性能也就很自然得到了提升。

此外，由于CPU对不同数据的处理方式不一样，就会造成不同类型的数据在各种运算处理如比较，排序等方面的处理效率存在差异。所以，对于我们需要经常进行比较计算以及排序等消耗CPU资源的字段，应该尽量选择处理更为迅速的字段类型。如通过整数类型代替浮点数或者字符类型。

9.3 规范的对象命名

规范的命名本身并不会对性能有任何影响，在这里单独列出一节来讲，主要是因为这是一个不太被人重视，但是对后期的数据库维护影响非常大的内容。就像编程语言各自的一些不成文编码基本规范一样，虽然在最初使用的时候并看不错太多的利益，反而会被人认为是一种束缚，但是当每一个人在接手维护一段编写很不规范的代码的时候，我估计大部分人都会非常郁闷，甚至在心里暗骂当初的编写者。其实任何系统都一样，没有任何规范可循，完全一个天马行空的作风，只会给后人（甚至可能是自己）留下一个让人摸不着头脑的烂摊子，难以维护。

对于数据库对象的命名规范其实可以很简单，而且业界也并不存在一个严格的统一规定，只需要在一个公司内足够统一基本就可以了。

一般来说，我个人建议需要注意以下一些方面：

1、 数据库和表名应尽可能和所服务的业务模块名一致；

这样，在 DBA 维护相关数据库对象的时候，新开发人员程序开发过程中，相关技术（或非技术）人员整理业务逻辑和数据关系的时候，都能够非常容易理解其中的关系。

2、 服务于同一子模块的一类表尽量以子模块名（或部分单词）为前缀或后缀；

对同类功能的表增加前缀或者后缀，也是让查看使用该表的各类人员能够很快的根据相关对象的名称就联想到相应的功能，以及相关业务。不论是从维护角度，还是从使用角度来看都会带来非常大的便利性。

3、 表名应尽量包含与所存放数据相对应的单词；

这对于新员工来说尤其重要，要想尽快的熟悉数据，尽快了解相关业务，快速的定位数据库中各表对应的数据意义是非常有帮助的。

4、 字段名称也尽量保持和实际数据相对应

这一点的意义我想各位读者朋友应该都非常的清楚，每个表都会有很多的字段对应数据的各种不同属性，要搞清楚各自代表的含义，除了完整规范的说明文档之外，命名清晰合理的字段名也是一个有用的补充，而且更为直接。

5、 索引名称尽量包含所有的索引键字段名或者缩写，且各字段名在索引名中的顺序应与索引键在索引中的索引顺序一致，且尽量包含一个类似于 idx 或者 ind 之类的前缀或者后缀，以表名其对象类型是索引，同时还可以包含该索引所属表的名称；

这样做最大的好处在于 DBA 在维护过程中能够非常直接清晰的通过索引名称就了解到该索引大部分的信息。

6、 约束等其他对象也应该尽可能包含所属表或其他对象的名称，以表名各自关系。

上面列出的只是一个比较初略的规范建议，各位读者朋友完全可以根据各自公司的习惯，制定自己的命名规范，只要适用，就可以了。规范不在多，而在实用。而且一旦制定的规范，就必须严格的按照规范执行，否则就变成了个花架子没有任何实际的意义了。

9.4 小结

通过这一章的内容，希望能够让大家明白一个道理，“数据库系统的性能不是优化出来的，更多的是设计出来的”。数据库 Schema 的设计并不如很多人想象的那样只是一个简单的对象对应实现，而是一个系统工程。要想设计出一个既性能高效又足够满足业务需求，既逻辑清晰又关系简单的数据库 Schema 结构，不仅仅需要足够的数据库系统知识，还需要足够了解应用系统的业务逻辑。

第 10 章 MySQL Server 性能优化

前言：

本章主要通过针对 MySQL Server（mysqld）相关实现机制的分析，得到一些相应的优化建议。主要涉及 MySQL 的安装以及相关参数设置的优化，但不包括 mysqld 之外的比如存储引擎相关的参数优化，存储引擎的相关参数设置建议将主要在下一章“常用存储引擎的优化”中进行说明。

10.1 MySQL 安装优化

选择合适的发行版本

1. 二进制发行版（包括RPM等包装好的特定二进制版本）

由于MySQL开源的特性，不仅仅MySQL AB提供了多个平台上面的多种二进制发行版本可以供大家选择，还有不少第三方公司（或者个人）也给我们提供了不少选择。

使用MySQL AB提供的二进制发行版本我们可以得到哪些好处？

a) 通过非常简单的安装方式快速完成MySQL的部署；
b) 安装版本是经过比较完善的功能和性能测试的编译版本；
c) 所使用的编译参数更具通用性的，且比较稳定；
d) 如果购买了MySQL的服务，将能最大程度的得到MySQL的技术支持；

第三方提供的MySQL发行版本大多是在MySQL AB官方提供的源代码方面做了或多或少的针对性改动，然后再编译而成。这些改动有些是在某些功能上面的改进，也有些是在某写操作的性能方面的改进。还有些由各OS厂商所提供的发行版本，则可能是在有些代码方面针对自己的OS做了一些相应的底层调用的调整，以使MySQL与自己的OS能够更完美的结合。当然，也有一些第三方发行版本并没有动过MySQL一行代码，仅仅只是在编译参数方面做了一些相关的调整，而让MySQL在某些特定场景下表现更优秀。

这样一说，听起来好像第三方发行的MySQL二进制版本要比MySQL AB官方提供的二进制发行版有更大的吸引力，那么我们是否就应该选用第三方提供的二进制发行版呢？先别着急，我们还需要进一步分析一下第三方发行版本可能存在哪些问题。

首先，由于第三方发行版本对MySQL所做的改动，很多都是为了应对发行者自己所处的特定场景而做出来的。所以，第三方发行版本并不一定适合其他所有使用者所处的环境。

其次，由于第三方发行版本的发行者并一定都是一个足够让人信任的公司（或者个人），在其生成自己的发行版本之前，是否有做过足够全面的功能和性能测试我们不得而知，在我们使用的时候是否会出现MySQL AB官方的发行版本中并不存在的bug？

最后，如果我们购买了MySQL 的相关服务，而又使用了第三方的发行版本，当我们的系统出现问题的时候，恐怕MySQL的支持工程师的支持工作会大打折扣，甚至可能会拒绝提供支持。

如果大家可以完全抛开以上这些可能存在隐患的顾虑，完全可以尝试使用非MySQL AB官方提供的二进制版本，而选用可能具有更多特性或者更高性能的发行版本了。

之前我也对网络上各种第三方二进制分发版本做过一些测试和比较，也发现了一些比较不错的版本，如Percona在整合了一些比较优秀的Patch之后的发行版本整体质量都还不错，使用者也比较多。当然，Percona不仅仅分发二进制版本，同时也分发整合了一些优秀Patch的源码包。对于希望使Percona提供的一些Patch的朋友，同时又希望能够自行编译以进一步优化和定制MySQL的朋友，也可以下载Percona提供的源码包。

对于二进制分发版本的安装，对于安装本身来说，我们基本上没有太多可以优化的地方，唯一可以做的就是当我们决定了选择第三方分发版本之后，可以根据自身环境和应用特点来选择适合我们环境的优化发行版本来安装。

2. 源码安装

与二进制发行版本相比，如果我们选择了通过源代码进行安装，那么在安装过程中我们能够对MySQL所做的调整将会更多更灵活一些。因为通过源代码编译我们可以：

a) 针对自己的硬件平台选用合适的编译器来优化编译后的二进制代码；
b) 根据不同的软件平台环境调整相关的编译参数；
c) 针对我们特定应用场景选择需要什么组件不需要什么组件；
d) 根据我们的所需要存储的数据内容选择只安装我们需要的字符集；
e) 同一台主机上面可以安装多个MySQL；
f) 等等其他一些可以根据特定应用场景所作的各种调整。

在源码安装给我们带来更大灵活性的同时，同样也给我们带来了可能引入的隐患：

a) 对编译参数的不够了解造成编译参数使用不当可能使编译出来的二进制代码不够稳定；
b) 对自己的应用环境把握失误而使用的优化参数可能反而使系统性能更差；
c) 还有一个并不能称之为隐患的小问题就是源码编译安装将使安装部署过程更为复杂，所花费的时间更长；

通过源码安装的最大特点就是可以让我们自行调整编译参数，最大程度的定制安装结果。下面我将自己在通过源码编译安装中的一些优化心得做一个简单的介绍，希望能够对大家有所帮助。

在通过源码安装的时候，最关键的一步就是配置编译参数，也就是执行通过configure命令所设定的各种编译选项。我们可以在MySQL源码所在的文件夹下面通过执行执行“./configure —help”得到可以设置的所有编译参数选项，如下：

```
`configure' configures this package to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...
    ... ...
Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
    ... ...
For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
    ... ...
Program names:
  --program-prefix=PREFIX       prepend PREFIX to installed program names
      ... ...
System types:
  --build=BUILD     configure for building on BUILD [guessed]
      ... ...
Optional Features:
  --disable-FEATURE  do not include FEATURE (same as --enable-FEATURE=no)
      ... ...
Optional Packages:
```

```
  --with-charset=CHARSET
      ... ...
  --without-innodb        Do not include the InnoDB table handler
      ... ...
Some influential environment variables:
  CC          C compiler command
      ... ...
  CCASFLAGS   assembler compiler flags (defaults to CFLAGS)
      ... ...
```

上面的输出内容中很多都已经省略了，大家完全可以通过自行测试得到更为丰富的内容输出。下面针对几个比较重要的编译参数做一个简单的介绍：

- "—prefix"：设定安装路径，默认为"/usr/local"；
- "—datadir"：设定 MySQL 数据文件存放路径；
- "—with-charset"：设定系统的默认字符集；
- "—with-collation"：系统默认的校验规则；
- "—with-extra-charsets"：出了默认字符集之外需要编译安装的字符集；
- "—with-unix-socket-path"：设定 socket 文件地址；
- "—with-tcp-port"：指定特定监听端口，默认为 3306；
- "—with-mysqld-user"：指定运行 mysqld 的 os 用户，默认为 mysql；
- "—without-query-cache"：禁用 Query Cache 功能；
- "—without-innodb"：禁用 Innodb 存储引擎；
- "--with-partition"：在 5.1 版本中开启 partition 支持特性；
- "--enable-thread-safe-client"：以线程方式编译客户端；
- "—with-pthread"：强制使用 pthread 线程库编译；
- "—with-named-thread-libs"：指定使用某个特定的线程库编译；
- "—without-debug"：使用非 debug 模式；
- "—with-mysqld-ldflags"：mysqld 的额外 link 参数；
- "—with-client-ldflags"：client 的额外 link 参数；
-

以上这些参数是在源码安装中比较常用的一些编译参数，其中前面几个编译参数主要是为了方便我们在安装的时候可以定制自己的系统，让系统更适合我们自己应用环境的相关规范，做到环境统一，并按照实际需求生成相应的二进制代码。而后面的一些参数主要是用来优化编译结果的。

我想大家应该都能理解一般来说，一个系统功能越复杂，其性能一般都会越差。所以，在我们安装编译 MySQL 的时候应该尽量只选用我们需要的组件，仅安装我们需要的存储引擎，仅编译我们需要的字符集，让我们的系统能够尽可能的简单，因为这样的 MySQL 也会给我们带来尽可能高的性能。

此外，对于一些特定的软件环境上，可能会有多种线程库的选择的，如果你对各个线程库较为了解，完全可以通过编译参数设定让MySQL使用最合适的线程库，让MySQL在我们特定的环境中发挥他最优化的一面。

源码包的编译参数中默认会以Debug模式生成二进制代码，而Debug模式给MySQL带来的性能损失是比较大的，所以当我们编译准备安装的产品代码的时候，一定不要忘记使用“—without-debug”参数禁用Debug模式。

而“—with-mysqld-ldflags”和“—with-client-ldflags”两个编译参数如果设置为“-all-static”的话，可以告诉编译器以静态方式编译来使编译结果代码得到最高的性能。使用静态编译和动态方式编译的代码相比，性能差距可能会达到5%到10%之多。

就我个人来说最常使用的编译配置参数如下，各位可以参照自行增删相关内容：

```
./configure --prefix=/usr/local/mysql \
--without-debug \
--without-bench \
--enable-thread-safe-client \
--enable-assembler \
--enable-profiling \
--with-mysqld-ldflags=-all-static \
--with-client-ldflags=-all-static \
--with-charset=latin1 \
--with-extra-charset=utf8,gbk \
--with-innodb \
--with-csv-storage-engine \
--with-federated-storage-engine \
--with-mysqld-user=mysql \
--without-embedded-server \
--with-server-suffix=-community \
--with-unix-socket-path=/usr/local/mysql/sock/mysql.sock
```

10.2 MySQL 日志设置优化

在安装完MySQL之后，肯定是需要对MySQL的各种参数选项进行一些优化调整的。虽然MySQL系统的伸缩性很强，既可以在有很充足的硬件资源环境下高效的运行，也可以在极少资源环境下很好的运行，但不管怎样，尽可能充足的硬件资源对MySQL的性能提升总是有帮助的。在这一节我们主要分析一下MySQL的日志（主要是Binlog）对系统性能的影响，并根据日志的相关特性得出相应的优化思路。

日志产生的性能影响

由于日志的记录带来的直接性能损耗就是数据库系统中最为昂贵的IO资源，所以对于日志的

在之前介绍MySQL物理架构的章节中，我们已经了解到了MySQL的日志包括错误日志（Error Log），更新日志（Update Log），二进制日志（Binlog），查询日志（Query Log），慢查询日志（Slow Query Log）等。当然，更新日志是老版本的MySQL才有的，目前已经被二进制日志替代。

在默认情况下，系统仅仅打开错误日志，关闭了其他所有日志，以达到尽可能减少IO损耗提高系统

性能的目的。但是在一般稍微重要一点的实际应用场景中，都至少需要打开二进制日志，因为这是MySQL很多存储引擎进行增量备份的基础，也是 MySQL 实现复制的基本条件。有时候为了进一步的性能优化，定位执行较慢的 SQL 语句，很多系统也会打开慢查询日志来记录执行时间超过特定数值（由我们自行设置）的 SQL 语句。

一般情况下，在生产系统中很少有系统会打开查询日志。因为查询日志打开之后会将 MySQL 中执行的每一条 Query 都记录到日志中，会该系统带来比较大的 IO 负担，而带来的实际效益却并不是非常大。一般只有在开发测试环境中，为了定位某些功能具体使用了哪些 SQL 语句的时候，才会在短时间段内打开该日志来做相应的分析。所以，在MySQL系统中，会对性能产生影响的MySQL日志（不包括各存储引擎自己的日志）主要就是 Binlog 了。

Binlog 相关参数及优化策略

我们首先看看Binlog的相关参数，通过执行如下命令可以获得关于Binlog的相关参数。当然，其中也显示出了“ innodb_locks_unsafe_for_binlog”这个 Innodb 存储引擎特有的与 Binlog 相关的参数：

```
mysql> show variables like '%binlog%';
+--------------------------------+------------+
| Variable_name                  | Value      |
+--------------------------------+------------+
| binlog_cache_size              | 1048576    |
| innodb_locks_unsafe_for_binlog | OFF        |
| max_binlog_cache_size          | 4294967295 |
| max_binlog_size                | 1073741824 |
| sync_binlog                    | 0          |
+--------------------------------+------------+
```

“binlog_cache_size″：在事务过程中容纳二进制日志 SQL 语句的缓存大小。二进制日志缓存是服务器支持事务存储引擎并且服务器启用了二进制日志(—log-bin 选项)的前提下为每个客户端分配的内存，注意，是每个Client都可以分配设置大小的binlog cache空间。如果读者朋友的系统中经常会出现多语句事务的华，可以尝试增加该值的大小，以获得更有的性能。当然，我们可以通过 MySQL 的以下两个状态变量来判断当前的binlog_cache_size 的状况：Binlog_cache_use 和 Binlog_cache_disk_use。

“max_binlog_cache_size”：和″binlog_cache_size″相对应，但是所代表的是binlog能够使用的最大cache内存大小。当我们执行多语句事务的时候，max_binlog_cache_size如果不够大的话，系统可能会报出“Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage”的错误。

“max_binlog_size”：Binlog日志最大值，一般来说设置为512M或者1G，但不能超过1G。该大小并不能非常严格控制Binlog大小，尤其是当到达Binlog比较靠近尾部而又遇到一个较大事务的时候，系统为了保证事务的完整性，不可能做切换日志的动作，只能将该事务的所有 SQL 都记录进入当前日志，直到该事务结束。这一点和Oracle的Redo日志有点不一样，因为Oracle的Redo日志所记录的是数据文件的物理位置的变化，而且里面同时记录了Redo和Undo相关的信息，所以同一个事务是否在一个日志中对 Oracle 来说并不关键。而 MySQL 在 Binlog 中所记录的是数据库逻辑变化信息，MySQL 称之为 Event，实际上就是带来数据库变化的DML之类的Query语句。

“sync_binlog”：这个参数是对于MySQL系统来说是至关重要的，他不仅影响到Binlog对MySQL所带来的性能损耗，而且还影响到MySQL中数据的完整性。对于“sync_binlog”参数的各种设置的说明如下：

- sync_binlog=0，当事务提交之后，MySQL不做fsync之类的磁盘同步指令刷新binlog_cache中的信息到磁盘，而让Filesystem自行决定什么时候来做同步，或者cache满了之后才同步到磁盘。
- sync_binlog=n，当每进行n次事务提交之后，MySQL将进行一次fsync之类的磁盘同步指令来将binlog_cache中的数据强制写入磁盘。

在MySQL中系统默认的设置是sync_binlog=0，也就是不做任何强制性的磁盘刷新指令，这时候的性能是最好的，但是风险也是最大的。因为一旦系统Crash，在binlog_cache中的所有binlog信息都会被丢失。而当设置为“1”的时候，是最安全但是性能损耗最大的设置。因为当设置为1的时候，即使系统Crash，也最多丢失binlog_cache中未完成的一个事务，对实际数据没有任何实质性影响。从以往经验和相关测试来看，对于高并发事务的系统来说，“sync_binlog”设置为0和设置为1的系统写入性能差距可能高达5倍甚至更多。

大家都知道，MySQL的复制（Replication），实际上就是通过将Master端的Binlog通过利用IO线程通过网络复制到Slave端，然后再通过SQL线程解析Binlog中的日志再应用到数据库中来实现的。所以，Binlog量的大小对IO线程以及Msater和Slave端之间的网络都会产生直接的影响。

MySQL中Binlog的产生量是没办法改变的，只要我们的Query改变了数据库中的数据，那么就必须将该Query所对应的Event记录到Binlog中。那我们是不是就没有办法优化复制了呢？当然不是，在MySQL复制环境中，实际上是是有8个参数可以让我们控制需要复制或者需要忽略而不进行复制的DB或者Table的，分别为：

- Binlog_Do_DB：设定哪些数据库（Schema）需要记录Binlog；
- Binlog_Ignore_DB：设定哪些数据库（Schema）不要记录Binlog；
- Replicate_Do_DB：设定需要复制的数据库（Schema），多个DB用逗号（“,”）分隔；
- Replicate_Ignore_DB：设定可以忽略的数据库（Schema）；
- Replicate_Do_Table：设定需要复制的Table；
- Replicate_Ignore_Table：设定可以忽略的Table；
- Replicate_Wild_Do_Table：功能同Replicate_Do_Table，但可以带通配符来进行设置；
- Replicate_Wild_Ignore_Table：功能同Replicate_Ignore_Table，可带通配符设置；

通过上面这八个参数，我们就可以非常方便按照实际需求，控制从Master端到Slave端的 Binlog量尽可能的少，从而减小Master端到Slave端的网络流量，减少IO线程的IO量，还能减少SQL线程的解析与应用SQL的数量，最终达到改善Slave上的数据延时问题。

实际上，上面这八个参数中的前面两个是设置在Master端的，而后面六个参数则是设置在Slave端的。虽然前面两个参数和后面六个参数在功能上并没有非常直接的关系，但是对于优化MySQL的Replication来说都可以启到相似的功能。当然也有一定的区别，其主要区别如下：

- 如果在Master端设置前面两个参数，不仅仅会让Master端的Binlog记录所带来的IO量减少，还会让Master端的IO线程就可以减少Binlog的读取量，传递给Slave端的IO线程的Binlog量自然就会较少。这样做的好处是可以减少网络IO，减少Slave端IO线程的IO量，减少Slave端的SQL线程的工作量，从而最大幅度的优化复制性能。当然，在Master端设置也存在一定的弊端，因为MySQL的判断是否需要复制某个Event不是根据产生该Event的Query所更改的数据

所在的DB，而是根据执行Query时刻所在的默认Schema，也就是我们登录时候指定的DB或者运行“USE DATABASE”中所指定的DB。只有当前默认DB和配置中所设定的DB完全吻合的时候 IO线程才会将该Event读取给Slave的IO线程。所以如果在系统中出现在默认DB和设定需要复制的DB不一样的情况下改变了需要复制的DB中某个Table的数据的时候，该Event是不会被复制到Slave中去的，这样就会造成Slave端的数据和Master的数据不一致的情况出现。同样，如果在默认Schema下更改了不需要复制的Schema中的数据，则会被复制到Slave端，当Slave端并没有该Schema的时候，则会造成复制出错而停止；

- 而如果是在Slave端设置后面的六个参数，在性能优化方面可能比在Master端要稍微逊色一点，因为不管是需要还是不需要复制的Event都被会被IO线程读取到Slave端，这样不仅仅增加了网络IO量，也给Slave端的IO线程增加了Relay Log的写入量。但是仍然可以减少Slave的SQL线程在Slave端的日志应用量。虽然性能方面稍有逊色，但是在Slave端设置复制过滤机制，可以保证不会出现因为默认Schema的问题而造成Slave和Master数据不一致或者复制出错的问题。

Slow Query Log 相关参数及使用建议

再来看看 Slow Query Log 的相关参数配置。有些时候，我们为了定位系统中效率比较地下的 Query 语句，则需要打开慢查询日志，也就是 Slow Query Log。我们可以如下查看系统慢查询日志的相关设置：

```
mysql> show variables like 'log_slow%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| log_slow_queries | ON    |
+------------------+-------+
1 row in set (0.00 sec)

mysql> show variables like 'long_query%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| long_query_time | 1     |
+-----------------+-------+
1 row in set (0.01 sec)
```

“log_slow_queries”参数显示了系统是否已经打开 Slow Query Log 功能，而“long_query_time”参数则告诉我们当前系统设置的 Slow Query 记录执行时间超过多长的 Query。在 MySQL AB 发行的 MySQL 版本中 Slow Query Log 可以设置的最短慢查询时间为 1 秒，这在有些时候可能没办法完全满足我们的要求，如果希望能够进一步缩短慢查询的时间限制，可以使用 Percona 提供的 microslow-patch（件成为 msl Patch）来突破该限制。msl patch 不仅仅能将慢查询时间减小到毫秒级别，同时还能通过一些特定的规则来过滤记录的 SQL，如仅记录涉及到某个表的 Slow Query 等等附加功能。考虑到篇幅问题，这里就不介绍 msl patch 给我们带来的更为详细的功能和使用，大家请参考官方介绍（http://www.mysqlperformanceblog.com/2008/04/20/updated-msl-microslow-patch-installation-walk-through/）

打开Slow Query Log功能对系统性能的整体影响没有Binlog那么大，毕竟Slow Query Log的数据量比较小，带来的IO损耗也就较小，但是，系统需要计算每一条Query的执行时间，所以消耗总是会有一些的，主要是CPU方面的消耗。如果大家的系统在CPU资源足够丰富的时候，可以不必在乎这一点点损耗，毕竟他可能会给我们带来更大性能优化的收获。但如果我们的 CPU资源也比较紧张的时候，也完全可以在大部分时候关闭该功能，而只需要间断性的打开Slow Query Log功能来定位可能存在的慢查询。

MySQL的其他日志由于使用很少（Query Log）或者性能影响很少，我们就不在此过多分析了，至于各个存储引擎相关的日志，我们留在后面“常用存储引擎优化”部分再做相应的分析。

10.3 Query Cache 优化

谈到Query Cache，恐怕使用过MySQL的大部分人都会或多或少有一些了解，因为在很多人看来他可以帮助我们将数据库的性能产生一个“质”的提升。但真的是这样吗？这一节我们就将如何合理的使用 MySQL 的Query Cache进行一些相应的分析并得出部分优化建议。

Query Cache真的是“尚方宝剑”吗？

MySQL 的 Query Cache实现原理实际上并不是特别的复杂，简单的来说就是将客户端请求的Query语句（当然仅限于SELECT类型的Query）通过一定的hash算法进行一个计算而得到一个hash值，存放在一个hash桶中。同时将该Query的结果集（Result Set）也存放在一个内存Cache中的。存放Query hash值的链表中的每一个hash值所在的节点中同时还存放了该Query所对应的 Result Set 的Cache所在的内存地址，以及该Query所涉及到的所有Table的标识等其他一些相关信息。系统接受到任何一个SELECT类型的Query的时候，首先计算出其hash值，然后通过该hash值到Query Cache中去匹配，如果找到了完全相同的Query，则直接将之前所Cache的 Result Set 返回给客户端而完全不需要进行后面的任何步骤即可完成这次请求。而后端的任何一个表的任何一条数据发生变化之后，也会通知 Query Cache，需要将所有与该Table有关的Query的Cache全部失效，并释放出之前占用的内存地址，以便后面其他的Query能够使用。

从上面的实现原理来看，Query Cache确实是以比较简单的实现带来巨大性能收益的功能。但是很多人可能都忽略了使用QueryCache之后所带来的负面影响：

a) Query语句的hash运算以及hash查找资源消耗。当我们使用Query Cache之后，每条SELECT类型的Query在到达MySQL之后，都需要进行一个hash运算然后查找是否存在该Query的Cache，虽然这个hash运算的算法可能已经非常高效了，hash查找的过程也已经足够的优化了，对于一条Query来说消耗的资源确实是非常非常的少，但是当我们每秒都有上千甚至几千条Query的时候，我们就不能对产生的CPU的消耗完全忽视了。
b) Query Cache的失效问题。如果我们的表变更比较频繁，则会造成Query Cache的失效率非常高。这里的表变更不仅仅指表中数据的变更，还包括结构或者索引等的任何变更。也就是说我们每次缓存到Query Cache中的Cache数据可能在刚存入后很快就会因为表中的数据被改变而被清除，然后新的相同Query进来之后无法使用到之前的Cache。
c) Query Cache中缓存的是 Result Set ，而不是数据页，也就是说，存在同一条记录被Cache多次的可能性存在。从而造成内存资源的过渡消耗。当然，可能有人会说我们可以限定 Query Cache的大小啊。是的，我们确实可以限定Query Cache的大小，但是这样，Query Cache就很容易造成因为内存不足而被换出，造成命中率的下降。

对于 Query Cache 的上面三个负面影响，如果单独拿出每一个影响来说都不会造成对整个系统多大的问题，并不会让大家对使用 Query Cache 产生太多顾虑。但是，当综合这三个负面影响一起考虑的话，恐怕 Query Cache 在很多人心目中就不再是以前的那把“尚方宝剑”了。

适度使用 Query Cache

虽然 Query Cache 的使用会存在一些负面影响，但是我们也应该相信其存在是必定有一定价值。我们完全不用因为 Query Cache 的上面三个负面影响就完全失去对 Query Cache 的信心。只要我们理解了 Query Cache 的实现原理，那么我们就完全可以通过一定的手段在使用 Query Cache 的时候扬长避短，重发发挥其优势，并有效的避开其劣势。

首先，我们需要根据 Query Cache 失效机制来判断哪些表适合使用 Query 哪些表不适合。由于 Query Cache 的失效主要是因为 Query 所依赖的 Table 的数据发生了变化，造成 Query 的 Result Set 可能已经有所改变而造成相关的 Query Cache 全部失效，那么我们就应该避免在查询变化频繁的 Table 的 Query 上使用，而应该在那些查询变化频率较小的 Table 的 Query 上面使用。MySQL 中针对 Query Cache 有两个专用的 SQL Hint（提示）：SQL_NO_CACHE 和 SQL_CACHE，分别代表强制不使用 Query Cache 和强制使用 Query Cache。我们完全可以利用这两个 SQL Hint，让 MySQL 知道我们希望哪些 SQL 使用 Query Cache 而哪些 SQL 就不要使用了。这样不仅可以让变化频繁 Table 的 Query 浪费 Query Cache 的内存，同时还可以减少 Query Cache 的检测量。

其次，对于那些变化非常小，大部分时候都是静态的数据，我们可以添加 SQL_CACHE 的 SQL Hint，强制 MySQL 使用 Query Cache，从而提高该表的查询性能。

最后，有些 SQL 的 Result Set 很大，如果使用 Query Cache 很容易造成 Cache 内存的不足，或者将之前一些老的 Cache 冲刷出去。对于这一类 Query 我们有两种方法可以解决，一是使用 SQL_NO_CACHE 参数来强制他不使用 Query Cache 而每次都直接从实际数据中去查找，另一种方法是通过设定“query_cache_limit”参数值来控制 Query Cache 中所 Cache 的最大 Result Set ，系统默认为 1M（1048576）。当某个 Query 的 Result Set 大于“query_cache_limit”所设定的值的时候，Query Cache 是不会 Cache 这个 Query 的。

Query Cache 的相关系统参数变量和状态变量

我们首先看看 Query Cache 的系统变量，可以通过执行如下命令获得 MySQL 中 Query Cache 相关的系统参数变量：

```
mysql> show variables like '%query_cache%';
+------------------------------+-----------+
| Variable_name                | Value     |
+------------------------------+-----------+
| have_query_cache             | YES       |
| query_cache_limit            | 1048576   |
| query_cache_min_res_unit     | 4096      |
| query_cache_size             | 268435456 |
| query_cache_type             | ON        |
```

```
| query_cache_wlock_invalidate | OFF        |
+------------------------------+-----------+
```

- “have_query_cache”：该 MySQL 是否支持 Query Cache；
- “query_cache_limit”：Query Cache 存放的单条 Query 最大 Result Set ，默认 1M；
- “query_cache_min_res_unit”：Query Cache 每个 Result Set 存放的最小内存大小，默认 4k；
- “query_cache_size”：系统中用于 Query Cache 内存的大小；
- “query_cache_type”：系统是否打开了 Query Cache 功能；
- “query_cache_wlock_invalidate”：针对于 MyISAM 存储引擎，设置当有 WRITE LOCK 在某个 Table 上面的时候，读请求是要等待 WRITE LOCK 释放资源之后再查询还是允许直接从 Query Cache 中读取结果，默认为 FALSE（可以直接从 Query Cache 中取得结果）。

以上参数的设置主要是“query_cache_limit”和“query_cache_min_res_unit”两个参数的设置需要做一些针对于应用的相关调整。如果我们需要 Cache 的 Result Set 一般都很小（小于 4k）的话，可以适当将“query_cache_min_res_unit”参数再调小一些，避免造成内存的浪费，“query_cache_limit”参数则不用调整。而如果我们需要Cache 的 Result Set 大部分都大于 4k 的话，则最好将“query_cache_min_res_unit”调整到和 Result Set 大小差不多，“query_cache_limit”的参数也应大于 Result Set 的大小。当然，可能有些时候我们比较难准确的估算 Result Set 的大小，那么当 Result Set 较大的时候，我们也并不是非得将“query_cache_min_res_unit”设置的和每个 Result Set 差不多大，是每个结果集的一半或者四分之一大小都可以，要想非常完美的完全不浪费任何内存确实也是不可能做到的。

如果我们要了解 Query Cache 的使用情况，则可以通过 Query Cache 相关的状态变量来获取，如通过如下命令：

```
mysql> show status like 'Qcache%';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| Qcache_free_blocks      | 7499       |
| Qcache_free_memory      | 190662000  |
| Qcache_hits             | 1888430018 |
| Qcache_inserts          | 1014096388 |
| Qcache_lowmem_prunes    | 106071885  |
| Qcache_not_cached       | 7951123988 |
| Qcache_queries_in_cache | 19315      |
| Qcache_total_blocks     | 47870      |
+-------------------------+------------+
```

- “Qcache_free_blocks”：Query Cache 中目前还有多少剩余的 blocks。如果该值显示较大，则说明 Query Cache 中的内存碎片较多了，可能需要寻找合适的机会进行整理（）。
- “Qcache_free_memory”：Query Cache 中目前剩余的内存大小。通过这个参数我们可以较为准确的观察出当前系统中的 Query Cache 内存大小是否足够，是需要增加还是过多了；
- “Qcache_hits”：多少次命中。通过这个参数我们可以查看到 Query Cache 的基本效果；
- “Qcache_inserts”：多少次未命中然后插入。通过“Qcache_hits”和“Qcache_inserts”两个参数我们就可以算出 Query Cache 的命中率了：

Query Cache 命中率 = Qcache_hits / (Qcache_hits + Qcache_inserts);

- "Qcache_lowmem_prunes"：多少条 Query 因为内存不足而被清除出 Query Cache。通过"Qcache_lowmem_prunes"和"Qcache_free_memory"相互结合，能够更清楚的了解到我们系统中Query Cache的内存大小是否真的足够，是否非常频繁的出现因为内存不足而有Query被换出
- "Qcache_not_cached"：因为query_cache_type的设置或者不能被cache的Query的数量；
- "Qcache_queries_in_cache"：当前Query Cache中cache的Query数量；
- "Qcache_total_blocks"：当前Query Cache中的block数量；

Query Cache的限制

Query Cache由于存放的都是逻辑结构的Result Set，而不是物理的数据页，所以在性能提升的同时，也会受到一些特定的限制。

a) 5.1.17之前的版本不能Cache 帮定变量的Query，但是从5.1.17版本开始，Query Cache已经开始支持帮定变量的Query了；
b) 所有子查询中的外部查询SQL不能被Cache；
c) 在Procedure，Function以及Trigger中的Query不能被Cache；
d) 包含其他很多每次执行可能得到不一样结果的函数的Query不能被Cache。

鉴于上面的这些限制，在使用Query Cache 的过程中，建议通过精确设置的方式来使用，仅仅让合适的表的数据可以进入 Query Cache，仅仅让某些 Query 的查询结果被 Cache。

10.4 MySQL Server 其他常用优化

除了安装，日志，Query Cache之外，可能影响MySQL Server整体性能的设置其他很多方面，如网络连接，线程管理，Table管理等。这一节我们将分析除了前面几节内容之外的可能影响MySQL Server性能的其他可优化的部分。

网络连接与连接线程

虽然MySQL的连接方式不仅仅只有通过网络方式，还可以通过命名管道的方式，但是不论是何种方式连接MySQL，在 MySQL 中都是通过线程的方式管理所有客户端请求的连接。每一个客户端连接都会有一个与之对应的生成一个连接线程。我们先看一下与网络连接的性能配置项及对性能的影响。

- max_conecctions：整个MySQL允许的最大连接数；
  这个参数主要影响的是整个MySQL应用的并发处理能力，当系统中实际需要的连接量大于max_conecctions的情况下，由于MySQL的设置限制，那么应用中必然会产生连接请求的等待，从而限制了相应的并发量。所以一般来说，只要MySQL主机性能允许，都是将该参数设置的尽可能大一点。一般来说500到800左右是一个比较合适的参考值

- max_user_connections：每个用户允许的最大连接数；
  上面的参数是限制了整个MySQL的连接数，而max_user_connections则是针对于单个用户的连接限制。在一般情况下我们可能都较少使用这个限制，只有在一些专门提供MySQL数据存储服务，或者是提供虚拟主机服务的应用中可能需要用到。除了限制的对象区别之外，其他方面和max_connections一样。这个参数的设置完全依赖于应用程序的连接用户数，对于普通的应用来

说，完全没有做太多的限制，可以尽量放开一些。

- net_buffer_length：网络包传输中，传输消息之前的 net buffer 初始化大小；
  这个参数主要可能影响的是网络传输的效率，由于该参数所设置的只是消息缓冲区的初始化大小，所以造成的影响主要是当我们的每次消息都很大的时候 MySQL 总是需要多次申请扩展该缓冲区大小。系统默认大小为 16KB，一般来说可以满足大多数场景，当然如果我们的查询都是非常小，每次网络传输量都很少，而且系统内存又比较紧缺的情况下，也可以适当将该值降低到 8KB。

- max_allowed_packet：在网络传输中，一次传消息输量的最大值；
  这个参数与 net_buffer_length 相对应，只不过是 net buffer 的最大值。当我们的消息传输量大于 net_buffer_length 的设置时，MySQL 会自动增大 net buffer 的大小，直到缓冲区大小达到 max_allowed_packet 所设置的值。系统默认值为 1MB，最大值是 1GB，必须设定为 1024 的倍数，单位为字节。

- back_log：在 MySQL 的连接请求等待队列中允许存放的最大连接请求数。
  连接请求等待队列，实际上是指当某一时刻客户端的连接请求数量过大的时候，MySQL 主线程没办法及时给每一个新的连接请求分配（或者创建）连接线程的时候，还没有分配到连接线程的所有请求将存放在一个等待队列中，这个队列就是 MySQL 的连接请求队列。当我们的系统存在瞬时的大量连接请求的时候，则应该注意 back_log 参数的设置。系统默认值为 50，最大可以设置为 65535。当我们增大 back_log 的设置的时候，同时还需要主义 OS 级别对网络监听队列的限制，因为如果 OS 的网络监听设置小于 MySQL 的 back_log 设置的时候，我们加大“back_log”设置是没有意义的。

上面介绍了网络连接交互相关的主要优化设置，下面我们再来看看与每一个客户端连接想对应的连接线程。

在 MySQL 中，为了尽可提高客户端请求创建连接这个过程的性能，实现了一个 Thread Cache 池，将空闲的连接线程存放在其中，而不是完成请求后就销毁。这样，当有新的连接请求的时候，MySQL 首先会检查 Thread Cache 池中是否存在空闲连接线程，如果存在则取出来直接使用，如果没有空闲连接线程，才创建新的连接线程。在 MySQL 中与连接线程相关的系统参数及状态变量说明如下：

- thread_cache_size：Thread Cache 池中应该存放的连接线程数。
  当系统最初启动的时候，并不会马上就创建 thread_cache_size 所设置数目的连接线程存放在 Thread Cache 池中，而是随着连接线程的创建及使用，慢慢的将用完的连接线程存入其中。当存放的连接线程达到 thread_cache_size 值之后，MySQL 就不会再续保存用完的连接线程了。

  如果我们的应用程序使用的短连接，Thread Cache 池的功效是最明显的。因为在短连接的数据库应用中，数据库连接的创建和销毁是非常频繁的，如果每次都需要让 MySQL 新建和销毁相应的连接线程，那么这个资源消耗实际上是非常大的，而当我们使用了 Thread Cache 之后，由于连接线程大部分都是在创建好了等待取用的状态，既不需要每次都重新创建，又不需要在使用完之后销毁，所以可以节省下大量的系统资源。所以在短连接的应用系统中，thread_cache_size 的值应该设置的相对大一些，不应该小于应用系统对数据库的实际并发请求数。

而如果我们使用的是长连接的时候，Thread Cache 的功效可能并没有使用短连接那样的大，但也并不是完全没有价值。因为应用程序即使是使用了长连接，也很难保证他们所管理的所有连接都能处于很稳定的状态，仍然会有不少连接关闭和新建的操作出现。在有些并发量较高，应用服务器数量较大的系统中，每分钟十來次的连接创建与关闭的操作是很常见的。而且如果应用服务器的连接池管理不是太好，容易产生连接池抖动的话，所产生的连接创建和销毁操作将会更多。所以即使是在使用长连接的应用环境中，Thread Cache 机制的利用仍然是对性能大有帮助的。只不过在长连接的环境中我们不需要将 thread_cache_size 参数设置太大，一般来说可能 50 到 100 之间应该就可以了。

● thread_stack：每个连接线程被创建的时候，MySQL 给他分配的内存大小。
  当 MySQL 创建一个新的连接线程的时候，是需要给他分配一定大小的内存堆栈空间，以便存放客户端的请求 Query 以及自身的各种状态和处理信息。不过一般来说如果不是对 MySQL 的连接线程处理机制十分熟悉的话，不应该轻易调整该参数的大小，使用系统的默认值（192KB）基本上可以所有的普通应用环境。如果该值设置太小，会影响 MySQL 连接线程能够处理客户端请求的 Query 内容的大小，以及用户创建的 Procedures 和 Functions 等。

上面介绍的这些都是我们可以怎样配置网络连接交互以及连接线程的性能相关参数，下面我们再看看该怎样检验上面所做的设置是否合理，是否有需要调整的地方。我们可以通过在系统中执行如下的几个命令来获得相关的状态信息来帮助大家检验设置的合理性：

我们现看看连接线程相关的系统变量的设置值：

```
mysql> show variables like 'thread%';
+-------------------+--------+
| Variable_name     | Value  |
+-------------------+--------+
| thread_cache_size | 64     |
| thread_stack      | 196608 |
+-------------------+--------+
```

再来看一下系统被连接的次数以及当前系统中连接线程的状态值：

```
mysql> show status like 'connections';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Connections   | 127   |
+---------------+-------+

mysql> show status like '%thread%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Delayed_insert_threads | 0     |
| Slow_launch_threads    | 0     |
| Threads_cached         | 4     |
```

```
| Threads_connected       | 7     |
| Threads_created         | 11    |
| Threads_running         | 1     |
+-------------------------+-------+
```

通过上面的命令，我们可以看出，系统设置了Thread Cache池最多将缓存32个连接线程，每个连接线程创建之初，系统分配192KB的内存堆栈空给他。系统启动到现在共接收到客户端的连接127次，共创建了11个连接线程，但前有7个连接线程处于和客户端连接的状态，而7个连接状态的线程中只有一个是active状态，也就是说只有一个正在处理客户端提交的俄请求。而在Thread Cache池中当共Cache了4个连接线程。

通过系统设置和当前状态的分析，我们可以发现，thread_cache_size的设置已经足够了，甚至还远大于系统的需要。所以我们可以适当减少 thread_cache_size 的设置，比如设置为 8 或者 16。根据Connections和Threads_created这两个系统状态值，我们还可以计算出系统新建连接连接的 Thread Cache命中率，也就是通过 Thread Cache池中取得连接线程的次数与系统接收的总连接次数的比率，如下：

Threads_Cache_Hit = (Connections - Threads_created) / Connections * 100%

我们可以通过上面的这个运算公式计算一下上面环境中的Thread Cache命中率：Thread_Cache_Hit = (127 - 12) / 127 * 100% = 90.55%

一般来说，当系统稳定运行一段时间之后，我们的 Thread Cache 命中率应该保持在 90%左右甚至更高的比率才算正常。可以看出上面环境中的Thread Cache命中比率基本还算是正常的。

Table Cache相关的优化

我们先来看一下 MySQL 打开表的相关机制。由于多线程的实现机制，为了尽可能的提高性能，在MySQL中每个线程都是独立的打开自己需要的表的文件描述符，而不是通过共享已经打开的表的文件描述符的机制来实现。当然，针对于不同的存储引擎可能有不同的处理方式。如MyISAM表，每一个客户端线程打开任何一个MyISAM表的数据文件都需要打开一个文件描述符，但如果是索引文件，则可以多个线程共享同一个索引文件的描述符。对于 Innodb 的存储引擎，如果我们使用的是共享表空间来存储数据，那么我们需要打开的文件描述符就比较少，而如果我们使用的是独享表空间方式来存储数据，则同样，由于存储表数据的数据文件较多，则同样会打开很多的表文件描述符。除了数据库的实际表或者索引打开以外，临时文件同样也需要使用文件描述符，同样会占用系统中open_files_limit的设置限额。

为了解决打开表文件描述符太过频繁的问题，MySQL在系统中实现了一个Table Cache的机制，和前面介绍的Thread Cache机制有点类似，主要就是Cache打开的所有表文件的描述符，当有新的请求的时候不需要再重新打开，使用结束的时候也不用立即关闭。通过这样的方式来减少因为频繁打开关闭文件描述符所带来的资源消耗。我们先看一看Table Cache相关的系统参数及状态变量。

在MySQL中我们通过table_cache（从MySQL5.1.3开始改为table_open_cache），来设置系统中为我们Cache的打开表文件描述符的数量。通过MySQL官方手册中的介绍，我们设置table_cache大小的时候应该通过max_connections参数计算得来，公式如下：

table_cache = max_connections * N;

其中N代表单个Query语句中所包含的最多Table的数量。但是我个人理解这样的计算其实并不是太准确，分析如下：

首先，max_connections是系统同时可以接受的最大连接数，但是这些连接并不一定都是active状态的，也就是说可能里面有不少连接都是处于Sleep状态。而处于Sleep状态的连接是不可能打开任何Table的。

其次，这个N为执行Query中包含最多的Table的Query所包含的Table的个数也并不是太合适，因为我们不能忽略索引文件的打开。虽然索引文件在各个连接线程之间是可以共享打开的连接描述符的，但总还是需要的。而且，如果我Query中的每个表的访问都是通过现通过索引定位检索的，甚至可能还是通过多个索引，那么该Query的执行所需要打开的文件描述符就更多了，可能是N的两倍甚至三倍。

最后，这个计算的公式只能计算出我们同一时刻需要打开的描述符的最大数量，而table_cache的设置也不一定非得根据这个极限值来设定，因为table_cache所设定的只是Cache打开的描述符的数量的大小，而不是最多能够打开的量的大小。

当然，上面的这些只是我个人的理解，也可能并不是太严谨，各位读者朋友如果觉得有其他的理解完全可以提出来大家再探讨。

我们可以通过如下方式查看table_cache的设置和当前系统中的使用状况：

```
mysql> show variables like 'table_cache';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| table_cache   | 512   |
+---------------+-------+

mysql> show status like 'open_tables';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 6     |
+---------------+-------+
```

上面的结果显示系统设置的table_cache为512个，也就是说在该MySQL中，Table Cache中可以Cache 512个打开文件的描述符；当前系统中打开的描述符仅仅则只有6个。

那么Table Cache池中Cache的描述符在什么情况下会被关闭呢？一般来说主要有以下集中情况会出现被Cache的描述符被关闭：

a) Table Cache的Cache池满了，而某个连接线程需要打开某个不在Table Cache中的表时，MySQL会通过一定的算法关闭某些没有在使用中的描述符；
b) 当我们执行Flush Table等命令的时候，MySQL会关闭当前Table Cache中Cache的所有文件描述符；
c) 当Table Cache中Cache的量超过table_cache参数设置的值的时候；

Sort Buffer，Join Buffer 和 Read Buffer

在 MySQL 中，之前介绍的多种 Cache 之外，还有在 Query 执行过程中的两种 Buffer 会对数据库的整体性能产生影响。

```
mysql> show variables like '%buffer%';
+-------------------------------+----------+
| Variable_name                 | Value    |
+-------------------------------+----------+
    ... ...
| join_buffer_size              | 4190208  |
    ... ...
| sort_buffer_size              | 2097144  |
+-------------------------------+----------+
```

- join_buffer_size：当我们的 Join 是 ALL，index，rang 或者 index_merge 的时候使用的 Buffer；
  实际上这种 Join 被称为 Full Join。实际上参与 Join 的每一个表都需要一个 Join Buffer，所以在 Join 出现的时候，至少是两个。Join Buffer 的设置在 MySQL 5.1.23 版本之前最大为 4GB，但是从 5.1.23 版本开始，在除了 Windows 之外的 64 位的平台上可以超出 4BG 的限制。系统默认是 128KB。

- sort_buffer_size：系统中对数据进行排序的时候使用的 Buffer；
  Sort Buffer 同样是针对单个 Thread 的，所以当多个 Thread 同时进行排序的时候，系统中就会出现多个 Sort Buffer。一般我们可以通过增大 Sort Buffer 的大小来提高 ORDER BY 或者是 GROUP BY 的处理性能。系统默认大小为 2MB，最大限制和 Join Buffer 一样，在 MySQL 5.1.23 版本之前最大为 4GB，从 5.1.23 版本开始，在除了 Windows 之外的 64 位的平台上可以超出 4GB 的限制。

如果应用系统中很少有 Join 语句出现，则可以不用太在乎 join_buffer_size 参数的大小设置，但是如果 Join 语句不是很少的话，个人建议可以适当增大 join_buffer_size 的设置到 1MB 左右，如果内存充足甚至可以设置为 2MB。对于 sort_buffer_size 参数来说，一般设置为 2MB 到 4MB 之间可以满足大多数应用的需求。当然，如果应用系统中的排序都比较大，内存充足且并发量不是特别的大的时候，也可以继续增大 sort_buffer_size 的设置。在这两个 Buffer 设置的时候，最需要注意的就是不要忘记是每个 Thread 都会创建自己独立的 Buffer，而不是整个系统共享的 Buffer，不要因为设置过大而造成系统内存不足。

10.5 小结

通过参数设置来进行性能优化所能够带来的性能提升可能并不会如很多人想象的那样产生质的飞跃，除非是之前的设置存在严重的不合理情况。我们不能将性能调优完全依托在通过 DBA 在数据库上线后的参数调整之上，而应该在系统设计和开发阶段就尽可能减少性能问题。当然，也不能否认参数调整在某些场景下对系统性能的影响比较大，但毕竟只是少数的特殊情况。

第 11 章 常用存储引擎优化

前言：

MySQL 提供的非常丰富的存储引擎种类供大家选择，有多种选择固然是好事，但是需要我们理解掌握的知识也会增加很多。每一种存储引擎都有各自的特长，也都存在一定的短处。如何将各种存储引擎在自己的应用环境中结合使用，扬长避短，也是一门不太简单的学问。本章选择最为常用的两种存储引擎进行针对性的优化建议，希望能够对读者朋友有一定的帮助。

11.1 MyISAM存储引擎优化

我们知道，MyISAM 存储引擎是 MySQL 最为古老的存储引擎之一，也是最为流行的存储引擎之一。对于以读请求为主的非事务系统来说，MyISAM 存储引擎由于其优异的性能表现及便利的维护管理方式无疑是大家最优先考虑的对象。这一节我们将通过分析 MyISAM 存储引擎的相关特性，来寻找提高 MyISAM 存储引擎性能的优化策略。

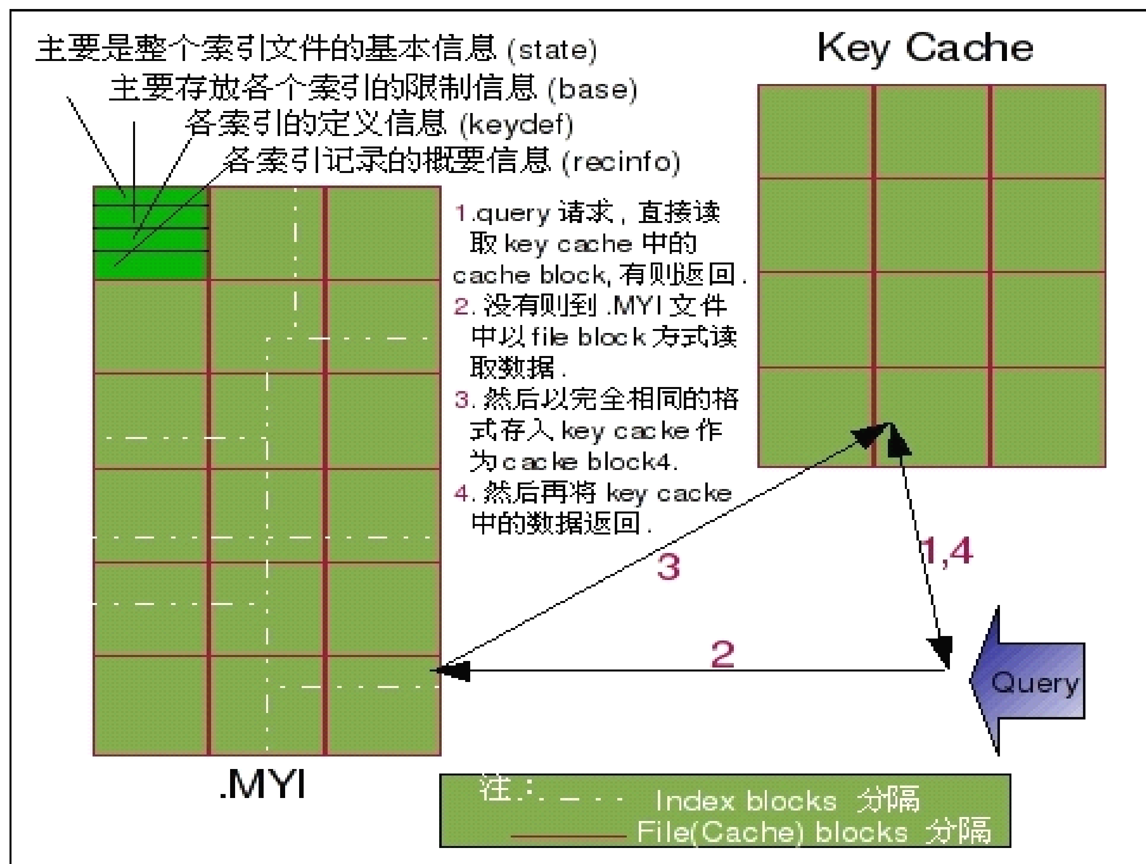
索引缓存优化

MyISAM 存储引擎的缓存策略是其和很多其他数据库乃至 MySQL 数据库的很多其他存储引擎不太一样的最大特性。因为他仅仅缓存索引数据，并不会缓存实际的表数据信息到内存中，而是将这一工作交给了 OS 级别的文件系统缓存。　所以，在数据库优化中非常重要的优化环节之一“缓存优化”的工作在使用 MyISAM 存储引擎的数据库的情况下，就完全集中在对索引缓存的优化上面了。

在分析优化索引缓存策略之前，我们先大概了解一下 MyISAM 存储引擎的索引实现机制以及索引文件的存放格式。

MyISAM 存储引擎的索引和数据是分开存放于“.MYI”文件中，每个“.MYI”文件由文件头和实际的索引数据。“.MYI”的文件头中主要存放四部分信息，分别称为：state（主要是整个索引文件的基本信息），base（各个索引的相关信息，主要是索引的限制信息），keydef（每个索引的定义信息）和 recinfo（每个索引记录的相关信息）。在文件头后面紧接着的就是实际的索引数据信息了。索引数据以 Block（Page）为最小单位，每个 block 中只会存在同一个索引的数据，这主要是基于提高索引的连续读性能的目的。在 MySQL 中，索引文件中索引数据的 block 被称为 Index Block，每个 Index Block 的大小并不一定相等。

在“.MYI”中，Index Block 的组织形式实际上只是一种逻辑上的，并不是物理意义上的。在物理上，实际上是以 File Block 的形式来存放在磁盘上面的。在 Key Cache 中缓存的索引信息是以“Cache Block”的形式组织存放的，“Cache Block”是相同大小的，和“.MYI”文件物理存储的Block（File Block）一样。在一条 Query 通过索引检索表数据的时候，首先会检查索引缓存（key_buffer_cache）中是否已经有需要的索引信息，如果没有，则会读取“.MYI”文件，将相应的索引数据读入 Key Cache 中的内存空间中，同样也是以 Block 形式存放，被称为 Cache Block。不过，数据的读入并不是以 Index Block 的形式来读入，而是以 File Block 的形式来读入的。以 File Block 形式读入到 Key Cache 之后的 Cache Block 实际上是于 File Block 完全一样的。如下图所示：



当我们从“.MYI”文件中读入 File Block 到 Key Cache 中 Cache Block 时候，如果整个 Key Cache 中已经没有空闲的 Cache Block 可以使用的话，将会通过 MySQL 实现的 LRU 相关算法将某些 Cache Block 清除出去，让新进来的 File Block 有地方呆。

我们先来分析一下与MyISAM索引缓存相关的几个系统参数和状态参数：

- key_buffer_size，索引缓存大小；

  这个参数用来设置整个MySQL中的常规 Key Cache大小。一般来说，如果我们的MySQL是运行在32位平台纸上，此值建议不要超过2GB大小。如果是运行在64位平台纸上则不用考虑此限制，但也最好不要超过4GB。

- key_buffer_block_size，索引缓存中的Cache Block Size；

  在前面我们已经介绍了，在Key Cache中的所有数据都是以Cache Block的形式存在，而key_buffer_block_size就是设置每个Cache Block的大小，实际上也同时限定了我们将“.MYI”文件中的Index Block被读入时候的File Block的大小。

- key_cache_division_limit，LRU链表中的Hot Area和Warm Area分界值；

  实际上，在MySQL的Key Cache中所使用的LRU算法并不像传统的算法一样仅仅只是通过访问频率以及最后访问时间来通过一个唯一的链表实现，而是将其分成了两部分。一部分用来存放使用比较频繁的Hot Cacke Lock（Hot Chain），被成为Hot Area，另外一部分则用来存放使用不是太频繁的Warm Cache Block（Warm Chain），被成为Warm Area。这样做的目的主要是为了保护使用比较频繁的Cache Block更不容易被换出。而key_cache_division_limit参数则是告诉MySQL该如何划分整个Cache Chain划分为Hot Chain和Warm Chain两部分，参数值为Warm Chain占整个Chain的百分比值。设置范围1～100，系统默认为100，也就是只有Warm Chain。

- key_cache_age_threshold，控制Cache Block从Hot Area降到Warm Area的限制；

  key_cache_age_threshold参数控制Hot Area中的Cache Block何时该被降级到Warm Area中。系统默认值为300，最小可以设置为100。值越小，被降级的可能性越大。

通过以上参数的合理设置，我们基本上可以完成MyISAM整体优化的70%的工作。但是如何的合理设置这些参数却不是一个很容易的事情。尤其是key_cache_division_limit和key_cache_age_threshold这两个参数的合理使用。

对于key_buffer_size的设置我们一般需要通过三个指标来计算，第一个是系统索引的总大小，第二个是系统可用物理内存，第三个是根据系统当前的Key Cache命中率。对于一个完全从零开始的全新系统的话，可能出了第二点可以拿到很清楚的数据之外，其他的两个数据都比较难获取，第三点是完全没有。当然，我们可以通过MySQL官方手册中给出的一个计算公式粗略的估算一下我们系统将来的索引大小，不过前提是要知道我们会创建哪些索引，然后通过各索引估算出索引键的长度，以及表中存放数据的条数，公式如下：

Key_Size = key_number * (key_length+4)/0.67

Max_key_buffer_size < Max_RAM - QCache_Usage - Threads_Usage - System_Usage

Threads_Usage = max_connections * (sort_buffer_size + join_buffer_size + read_buffer_size + read_rnd_buffer_size + thread_stack)

当然，考虑到活跃数据的问题，我们并不需要将key_buffer_size设置到可以将所有的索引都放下的大小，这时候我们就需要Key Cache的命中率数据来帮忙了。下面我们再来看一下系统中记录的与Key

Cache 相关的性能状态参数变量。

- Key_blocks_not_flushed，已经更改但还未刷新到磁盘的 Dirty Cache Block；
- Key_blocks_unused，目前未被使用的 Cache Block 数目；
- Key_blocks_used，已经使用了的 Cache Block 数目；
- Key_read_requests，Cache Block 被请求读取的总次数；
- Key_reads，在 Cache Block 中找不到需要读取的 Key 信息后到“.MYI”文件中读取的次数；
- Key_write_requests，Cache Block 被请求修改的总次数；
- Key_writes，在 Cache Block 中找不到需要修改的 Key 信息后到“.MYI”文件中读入再修改的次数；

由于上面各个状态参数在 MySQL 官方文档中都有较为详细的描述，所以上面仅做基本的说明。当我们的系统上线之后，我们就可以通过上面这些状态参数的状态值得到系统当前的 Key Cache 使用的详细情况和性能状态。

Key_buffer_UsageRatio = (1 - Key_blocks_used/(Key_blocks_used + Key_blocks_unused)) * 100%

Key_Buffer_Read_HitRatio = (1 - Key_reads/Key_read_requests) * 100%

Key_Buffer_Write_HitRatio = (1 - Key_writes/Key_Write_requests) * 100%

通过上面的这三个比率数据，就可以很清楚的知道我们的 Key Cache 设置是否合理，尤其是 Key_Buffer_Read_HitRatio 参数和 Key_buffer_UsageRatio 这两个比率。一般来说 Key_buffer_UsageRatio 应该在 99%以上甚至 100%，如果该值过低，则说明我们的 key_buffer_size 设置过大，MySQL 根本使用不完。Key_Buffer_Read_HitRatio 也应该尽可能的高。如果该值较低，则很有可能是我们的 key_buffer_size 设置过小，需要适当增加 key_buffer_size 值，也有可能是 key_cache_age_threshold 和 key_cache_division_limit 的设置不当，造成 Key Cache cache 失效太快。一般来说，在实际应用场景中，很少有人调整 key_cache_age_threshold 和 key_cache_division_limit 这两个参数的值，大都是使用系统的默认值。

多 Key Cache 的使用

从 MySQL4.1.1 版本开始，MyISAM 开始支持多个 Key Cache 并存的的功能。也就是说我们可以根据不同的需要设置多个 Key Cache 了，如将使用非常频繁而且基本不会被更新的表放入一个 Key Cache 中以防止在公共 Key Cache 中被清除出去，而那些使用并不是很频繁而且可能会经常被更新的 Key 放入另外一个 Key Cache 中。这样就可以避免出现某些场景下大批量的 Key 被读入 Key Cache 的时候，因为 Key Cache 空间问题使本来命中率很高的 Key 也不得不被清除出去。

MySQL 官方建议在比较繁忙的系统上一般可以设置三个 Key Cache：
一个 Hot Cache 使用 20%的大小用来存放使用非常频繁且更新很少的表的索引；
一个 Cold Cache 使用 20%的大小用来存放更新很频繁的表的索引；
一个 Warm Cache 使用剩下的 60%空间，作为整个系统默认的 Key Cache；

多个 Key Cache 的具体使用方法在 MySQL 官方手册中有比较详细的介绍，这里就不再累述了，有兴趣

的读者朋友可以自行查阅研究。

Key Cache 的 Mutex 问题

MySQL 索引缓存是所有线程共享的全局缓存，当多线程同时并发读取某一个 Cache Block 的时候并不会有任何问题，每个线程都可以同时读取该 Cache Block。但是当某个 Cache Block 正在被一个线程更新或者读入的时候，则该线程就会通过 mutex 锁定该 Cache Block 以达到不允许其他线程再同时更新或者读取。所以在高并发的环境下，如果 Key Cache 大小不够充足是非常容易因为 Cache Block 的 Mutex 问题造成严重的性能影响。而且在目前正式发行的所有 MySQL 版本中，Mutex 的处理机制存在一定的问题，使得当我们的 Active 线程数量稍微高一些的时候，就非常容易出现 Cache Block 的 Mutex 问题，甚至有人将此性能问题作为 Bug（#31551）报告给了 MySQL AB。

Key Cache 预加载

在 MySQL 中，为了让系统刚启动之后不至于因为 Cache 中没有任何数据而出现短时间的负载过高或者是响应不够及时的问题。MySQL 提供了 Key Cache 预加载功能，可以通过相关命令（LOAD INDEX INTO CACHE tb_name_list ...），将指定表的所有索引都加载到内存中，而且还可以通过相关参数控制是否只 Load 根结点和枝节点还是将页节点也全部 Load 进来，主要是为 Key Cache 的容量考虑。

对于这种启动后立即加载的操作，可以利用 MySQL 的 init_file 参数来设置相关的命令，如下：

```
mysql@sky:~$ cat /usr/local/mysql/etc/init.sql
SET GLOBAL hot_cache.key_buffer_size=16777216
SET GLOBAL cold_cache.key_buffer_size=16777216
CACHE INDEX example.top_message in hot_cache
CACHE INDEX example.event in cold_cache
LOAD INDEX INTO CACHE example.top_message,example.event IGNORE LEAVES
LOAD INDEX INTO CACHE example.user IGNORE LEAVES,exmple.groups
```

这里我的 init file 中首先设置了两个 Key Cache（hot cache 和 cold cache）各为 16M，然后分别将 top_message 这个变动很少的表的索引 Cache 到 Hot Cache，再将 event 这个变动非常频繁的表的索引 Cache 到了 Cold Cache 中，最后再通过 LOAD INDEX INTO CACHE 命令预加载了 top_message，groups 这两个表所有索引的所有节点以及 event 和 user 这两个表索引的非叶子节点数据到 Key Cache 中，以提高系统启动之初的响应能力。

NULL 值对统计信息的影响

虽然都是使用 B-Tree 索引，但是 MyISAM 索引和 Oracle 索引的处理方式不太一样，MyISAM 的索引中是会记录值为 NULL 的列信息的，只不过 NULL 值的索引键占用的空间非常少。所以，NULL 值的处理方式可能会影响到 MySQL 的查询优化器对执行计划的选择。所以 MySQL 就给我们提供了 myisam_stats_method 这个参数让我们可以自行决定对索引中的 NULL 值的处理方式。

myisam_stats_method 参数的作用就是让我们告诉 MyISAM 在收集统计信息的时候，是认为所有 NULL 值都是等同还是认为每个 NULL 值都认为是完全不相等的值，所以其可设置的值也为 nulls_unequal 和 nulls_equal。

当我们设置 myisam_stats_method = nulls_unequal，MyISAM 在搜集统计信息的时候会认为每个 NULL 值都不同，则基于该字段的索引的 Cardinality 就会更大，也就是说 MyISAM 会认为 DISTINCT 值数

量更多，这样就会让查询优化器处理 Query 的时候使用该索引的倾向性更高。

而当我们设置 myisam_stats_method = nulls_equal 之后，MyISAM 搜集统计信息的时候则会认为每个 NULL 值的都是一样的，这样 Cardinality 数值会降低，优化器选择执行计划的时候放弃该索引的倾向性会更高。

当然，上面所说的都是相对于使用等值查询的时候，而且 NULL 值占比较大的情况下，如果我们的 NULL 值本身就很少，那不管我们是使用 nulls_unequal 还是 nulls_equal，对优化器选择执行计划的影响是很小很小的。

表读取缓存优化

在 MySQL 中有两种读取数据文件的缓冲区，一种是 Sequential Scan 方式（如全表扫描）扫描表数据的时候使用，另一种则是在 Random Scan（如通过索引扫描）的时候使用。虽然这两种文件读取缓冲区并不是 MyISAM 存储引擎所特有的，但是由于 MyISAM 存储引擎并不会 Cache 数据（.MYD）文件，每次对数据文件的访问都需要通过调用文件系统的相关指令从磁盘上面读取物理文件。所以，每次读取数据文件需要使用的内存缓冲区的设置就对数据文件访问的性能非常重要了。在 MySQL 中对应这两种缓冲区的相关参数如下：

- read_buffer_size，以 Sequential Scan 方式扫描表数据时候使用的 Buffer；
  每个 Thread 进行 Sequential Scan 的时候都会产生该 Buffer，所以在设置的时候尽量不要太高，避免因为并发太大造成内存不够。系统默认为 128KB，最大为 2GB，设置的值必须是 4KB 的倍数，否则系统会自动更改成小于设置值的最大的 4KB 的倍数。
  一般来说，可以尝试适当调大此参数看是否能够改善全表扫描的性能。在不同的平台上可能会有不同的表现，这主要与 OS 级别的文件系统 IO 大小有关。所以该参数的设置最好是在真实环境上面通过多次更改测试调整，才能选找到一个最佳值。

- read_rnd_buffer_size，进行 Random Scan 的时候使用的 Buffer；
  read_rnd_buffer_size 所设置的 Buffer 实际上刚好和 read_buffer_size 所设置的 Buffer 相反，一个是顺序读的时候使用，一个是随机读的时候使用。但是两者都是针对于线程的设置，每个线程都可能产生两种 Buffer 中的任何一个。read_rnd_buffer_size 的默认值 256KB，最大值为 4G。
  一般来说，read_rnd_buffer_size 值的适当调大，对提高 ORDER BY 操作的性能有一定的效果。

这两个读取缓冲区都是线程独享的，每个线程在需要的时候都会创建一个（或者两个）系统中设置大小的缓冲区，所以在设置上面两个参数的时候一定不要过于激进，而应该根须系统可能的最大连接数和系统可用内存大小，计算出最大可设置值。

并发优化

在查询方面，MyISAM 存储引擎的并发并没有太大的问题，而且性能也非常的高。而且如果觉得光靠 Key Cache 来缓存索引还是不够快的话，我们还可以通过 Query Cache 功能来直接缓存 Query 的结果集。

但是，由于 MyISAM 存储引擎的表级锁定机制，以及读写互斥的问题，其并发写的性能一直是一个让人比较头疼的问题。一般来说，我们能做的主要也就只有以下几点了：

1. 打开 concurrent_insert 的功能，提高 INSERT 操作和 SELECT 之间的并发处理，使二者尽可能并行。大部分情况下 concurrent_insert 的值都被设置为 1，当表中没有删除记录留下的空余空间的时候都可以在尾部并行插入。这其实也是 MyISAM 的默认设置。如果我们的系统主要以写为主，尤其是

有大量的 INSERT 的时候。为了尽可能提高 INSERT 的效率，我们可以将 concurrent_insert 设置为 2，也就是告诉 MyISAM，不管在表中是否有删除行留下的空余空间，都在尾部进行并发插入，使 INSERT 和 SELECT 能够互不干扰。
2. 控制写入操作的大小，尽量让每次写入操作都能够很快的完成，以防止时间过程的阻塞动作。
3. 通过牺牲读取效率来提高写入效率。为了尽可能让写入更快，可以适当调整读和写的优先级别，让写入操作的优先级高于读操作的优先级。

对于一个表级锁定的存储引擎来说，除了 concurrent_insert 这个比较特殊的特性之外，可以说基本上都只能是串行的写。所以虽然上面给出了三点建议，但是后面两点也只能算是优化建议，并不是真正意义上的并发优化建议。

其他可以优化的地方

除了上面我们分析的这几个方面之外，MyISAM 实际上还存在其他一些可以优化的地方和一些常用的优化技巧。

1. 通过 OPTIMIZE 命令来整理 MyISAM 表的文件。这就像我们使用 Windows 操作系统会每过一段时间后都会做一次磁盘碎片整理，让系统中的文件尽量使用连续空间，提高文件的访问速度。MyISAM 在通过 OPTIMIZE 优化整理的时候，主要也是将因为数据删除和更新造成的碎片空间清理，使整个文件连续在一起。一般来说，在每次做了较大的数据删除操作之后都需要做一次 OPTIMIZE 操作。而且每个季度都应该有一次 OPTIMIZE 的维护操作。
2. 设置 myisam_max_[extra]_sort_file_size 足够大，对 REPAIR TABLE 的效率可能会有较大改善。
3. 在执行 CREATE INDEX 或者 REPAIR TABLE 等需要大的排序操作的之前可以通过调整 session 级别的 myisam_sort_buffer_size 参数值来提高排序操作的效率。
4. 通过打开 delay_key_write 功能，减少 IO 同步的操作，提高写入性能。
5. 通过调整 bulk_insert_buffer_size 来提高 INSERT...SELECT...这样的 bulk insert 操作的整体性能，LOAD DATA INFILE...的性能也可以得到改善。当然，在设置此参数的时候，也不应该一味的追求很大，很多时候过渡追求极端反而会影响系统整体性能，毕竟系统性能是从整体来看的，而不能仅仅针对某一个或者某一类操作。

11.2 Innodb 存储引擎优化

Innodb 存储引擎和 MyISAM 存储引擎最大区别主要有四点，第一点是缓存机制，第二点是事务支持，第三点是锁定实现，最后一点就是数据存储方式的差异。在整体性能表现方面，Innodb 和 MyISAM 两个存储引擎在不同的场景下差异比较大，主要原因也正是因为上面这四个主要区别所造成的。锁定相关的优化我们已经在“MySQL 数据库锁定机制”一章中做过相关的分析了，所以，本节关于 Innodb 存储引擎优化的分析，也将主要从其他三个方面展开。

11.2.1 Innodb 缓存相关优化

无论是对于哪一种数据库来说，缓存技术都是提高数据库性能的关键技术，物理磁盘的访问速度永远都会与内存的访问速度永远都不是一个数量级的。通过缓存技术无论是在读还是写方面都可以大大提高数据库整体性能。

Innodb_buffer_pool_size 的合理设置

Innodb 存储引擎的缓存机制和 MyISAM 的最大区别就在于 Innodb 不仅仅缓存索引，同时还会缓存实际的数据。所以，完全相同的数据库，使用 Innodb 存储引擎可以使用更多的内存来缓存数据库相关的信息，当然前提是要有足够的物理内存。这对于在现在这个内存价格不断降低的时代，无疑是个很吸引人的特性。

innodb_buffer_pool_size 参数用来设置 Innodb 最主要的 Buffer(Innodb_Buffer_Pool)的大小，也就是缓存用户表及索引数据的最主要缓存空间，对 Innodb 整体性能影响也最大。无论是 MySQL 官方手册还是网络上很多人所分享的 Innodb 优化建议，都简单的建议将 Innodb 的 Buffer Pool 设置为整个系统物理内存的 50% ～ 80% 之间。如此轻率的给出此类建议，我个人觉得实在是有些不妥。

不管是多么简单的参数，都可能与实际运行场景有很大的关系。完全相同的设置，不同的场景下的表现可能相差很大。就从 Innodb 的 Buffer Pool 到底该设置多大这个问题来看，我们首先需要确定的是这台主机是不是就只提供 MySQL 服务？MySQL 需要提供的的最大连接数是多少？MySQL 中是否还有 MyISAM 等其他存储引擎提供服务？如果有，其他存储引擎所需要使用的 Cache 需要多大？

假设是一台单独给 MySQL 使用的主机，物理内存总大小为 8G，MySQL 最大连接数为 500，同时还使用了 MyISAM 存储引擎，这时候我们的整体内存该如何分配呢？

内存分配为如下几大部分：

a) 系统使用，假设预留 800M；

b) 线程独享，约 2GB = 500 * (1MB + 1MB + 1MB + 512KB + 512KB)，组成大概如下:
   sort_buffer_size：1MB
   join_buffer_size：1MB
   read_buffer_size：1MB
   read_rnd_buffer_size：512KB
   thread_statck：512KB

c) MyISAM Key Cache，假设大概为 1.5GB；

d) Innodb Buffer Pool 最大可用量：8GB - 800MB - 2GB - 1.5GB = 3.7GB；

假设这个时候我们还按照 50%～80%的建议来设置，最小也是 4GB，而通过上面的估算，最大可用值在 3.7GB 左右，那么很可能在系统负载很高当线程独享内存差不多出现极限情况的时候，系统很可能就会出现内存不足的问题了。而且上面还仅仅只是列出了一些使用内存较大的地方，如果进一步细化，很可能可用内存会更少。

上面只是一个简单的示例分析，实际情况并不一定是这样的，这里只是希望大家了解，在设置一些参数的时候，千万不要想当然，一定要详细的分析可能出现的情况，然后再通过不断测试调整来达到自己所处环境的最优配置。就我个人而言，正式环境上线之初，我一般都会采取相对保守的参数配置策略。上线之后，再根据实际情况和收集到的各种性能数据进行针对性的调整。

当系统上线之后，我们可以通过 Innodb 存储引擎提供给我们的关于 Buffer Pool 的实时状态信息作出进一步分析，来确定系统中 Innodb 的 Buffer Pool 使用情况是否正常高效：

```
sky@localhost : example 08:47:54> show status like 'Innodb_buffer_pool_%';
    +-----------------------------------+-------+
    | Variable_name                     | Value |
    +-----------------------------------+-------+
    | Innodb_buffer_pool_pages_data     | 70    |
    | Innodb_buffer_pool_pages_dirty    | 0     |
    | Innodb_buffer_pool_pages_flushed  | 0     |
    | Innodb_buffer_pool_pages_free     | 1978  |
    | Innodb_buffer_pool_pages_latched  | 0     |
    | Innodb_buffer_pool_pages_misc     | 0     |
    | Innodb_buffer_pool_pages_total    | 2048  |
    | Innodb_buffer_pool_read_ahead_rnd | 1     |
    | Innodb_buffer_pool_read_ahead_seq | 0     |
    | Innodb_buffer_pool_read_requests  | 329   |
    | Innodb_buffer_pool_reads          | 19    |
    | Innodb_buffer_pool_wait_free      | 0     |
    | Innodb_buffer_pool_write_requests | 0     |
    +-----------------------------------+-------+
```

从上面的值我们可以看出总共 2048 pages，还有 1978 是 Free 状态的仅仅只有 70 个 page 有数据，read 请求 329 次，其中有 19 次所请求的数据在 buffer pool 中没有，也就是说有 19 次是通过读取物理磁盘来读取数据的，所以很容易也就得出了 Innodb Buffer Pool 的 Read 命中率大概在为：(329 - 19) / 329 * 100% = 94.22%。

当然，通过上面的数据，我们还可以分析出 write 命中率，可以得到发生了多少次 read_ahead_rnd，多少次 read_ahead_seq，发生过多少次 latch，多少次因为 Buffer 空间大小不足而产生 wait_free 等等。

单从这里的数据来看，我们设置的 Buffer Pool 过大，仅仅使用 70 / 2048 * 100% = 3.4%。

在 Innodb Buffer Pool 中，还有一个非常重要的概念，叫做“预读”。一般来说，预读概念主要是在一些高端存储上面才会有，简单来说就是通过分析数据请求的特点来自动判断出客户在请求当前数据块之后可能会继续请求的数据快。通过该自动判断之后，存储引擎可能就会一次将当前请求的数据库和后面可能请求的下一个（或者几个）数据库一次全部读出，以期望通过这种方式减少磁盘 IO 次数提高 IO 性能。在上面列出的状态参数中就有两个专门针对预读：

Innodb_buffer_pool_read_ahead_rnd，记录进行随机读的时候产生的预读次数；

Innodb_buffer_pool_read_ahead_seq，记录连续读的时候产生的预读次数；

innodb_log_buffer_size 参数的使用

顾名思义，这个参数就是用来设置 Innodb 的 Log Buffer 大小的，系统默认值为 1MB。Log Buffer 的主要作用就是缓冲 Log 数据，提高写 Log 的 IO 性能。一般来说，如果你的系统不是写负载非常高且以大事务居多的话，8MB 以内的大小就完全足够了。

我们也可以通过系统状态参数提供的性能统计数据来分析 Log 的使用情况：

```
sky@localhost : example 10:11:05> show status like 'innodb_log%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| Innodb_log_waits          | 0     |
| Innodb_log_write_requests | 6     |
| Innodb_log_writes         | 2     |
+---------------------------+-------+
```

通过这三个状态参数我们可以很清楚的看到 Log Buffer 的等待次数等性能状态。

当然，如果完全从 Log Buffer 本身来说，自然是大一些会减少更多的磁盘 IO。但是由于 Log 本身是为了保护数据安全而产生的，而 Log 从 Buffer 到磁盘的刷新频率和控制数据安全一致的事务直接相关，并且也有相关参数来控制（innodb_flush_log_at_trx_commit），所以关于 Log 相关的更详细的实现机制和优化在后面的“事务优化”中再做更详细的分析，这里就不展开了。

innodb_additional_mem_pool_size 参数理解

innodb_additional_mem_pool_size 所设置的是用于存放 Innodb 的字典信息和其他一些内部结构所需要的内存空间。所以我们的 Innodb 表越多，所需要的空间自然也就越大，系统默认值仅有 1MB。当然，如果 Innodb 实际运行过程中出现了实际需要的内存比设置值更大的时候，Innodb 也会继续通过 OS 来申请内存空间，并且会在 MySQL 的错误日志中记录一条相应的警告信息让我们知晓。

从我个人的经验来看，一个常规的几百个 Innodb 表的 MySQL，如果不是每个表都是上百个字段的话，20MB 内存已经足够了。当然，如果你有足够多的内存，完全可以继续增大这个值的设置。实际上，innodb_additional_mem_pool_size 参数对系统整体性能并无太大的影响，所以只要能存放需要的数据即可，设置超过实际所需的内存并没有太大意义，只是浪费内存而已。

Double Write Buffer

Double Write Buffer 是 Innodb 所使用的一种较为独特的文件 Flush 实现技术，主要做用是为了通过减少文件同步次数提高 IO 性能的情况下，提高系统 Crash 或者断电情况下数据的安全性，避免写入的数据不完整。

一般来说，Innodb 在将数据同步到数据文件进行持久化之前，首先会将需要同步的内容写入存在于

表空间中的系统保留的存储空间，也就是被我们称之为 Double Write Buffer 的地方，然后再将数据进行文件同步。所以实质上，Double Write Buffer 中就是存放了一份需要同步到文件中数据的一个备份，以便在遇到系统 Crash 或者主机断电的时候，能够校验最后一次文件同步是否准确的完成了，如果未完成，则可以通过这个备份来继续完成工作，保证数据的正确性。

那这样 Innodb 不是又一次增加了整体 IO 量了吗？这样不是可能会影响系统的性能么？这个完全不用太担心，因为 Double Write Buffer 是一块连续的磁盘空间，所有写入 Double Write Buffer 的操作都是连续的顺序写入操作，与整个同步过程相比，这点 IO 消耗所占的比例是非常小的。为了保证数据的准确性，这样一点点性能损失是完全可以接受的。

实际上，并不是所有的场景都需要使用 Double Write 这样的机制来保证数据的安全准确性，比如当我们使用某些特别文件系统的时候，如在 Solaris 平台上非常著名的 ZFS 文件系统，他就可以自己保证文件写入的完整性。而且在我们的 Slave 端，也可以禁用 Double Write 机制。

Adaptive Hash Index

在 Innodb 中，实现了一个自动监测各表索引的变化情况的机制，然后通过一系列的算法来判定如果存在一个 Hash Index 是否会对索引搜索带来性能改善。如果 Innodb 认为可以通过 Hash Index 来提高检索效率，他就会在内部自己建立一个基于某个 B-Tree 索引的 Hash Index，而且会根据该 B-Tree 索引的变化自行调整，这就是我们常说的 Adaptive Hash Index。当然，Innodb 并不一定会将整个 B-Tree 索引完全的转换为 Hash Index，可能仅仅只是取用该 B-Tree 索引键一定长度的前缀来构造一个 Hash Index。

Adaptive Hash Index 并不会进行持久化存放在磁盘上面，仅仅存在于 Buffer Pool 中。所以，在每次 MySQL 刚启动之后是并不存在 Adaptive Hash Index 的，只有在停工服务之后，Innodb 才会根据相应的请求来构建。

Adaptive Hash Index 的目的并不是为了改善磁盘 IO 的性能，而是为了提高 Buffer Pool 中的数据的访问效率，说的更浅显一点就是给 Buffer Pool 中的数据做的索引。所以，Innodb 在具有大容量内存（可以设置大的 Buffer Pool）的主机上，对于其他存储引擎来说，会存在一定的性能优势。

11.2.2 事务优化

选择合适的事务隔离级别

Innodb 存储引擎是 MySQL 中少有的支持事务的存储引擎之一，这也是其成为目前 MySQL 环境中使用最广泛存储引擎之一的一个重要原因。由于事务隔离的实现本身是需要消耗大量的内存和计算资源，而且不同的隔离级别所消耗的资源也不一样，性能表现也各不相同。所以我们

首先我们大概了解一下 Innodb 所支持的各种事务隔离级别。通过 Innodb 的参考手册，我们得到 Innodb 在事务隔离级别方面支持的信息如下：

1. READ UNCOMMITTED

常被成为 Dirty Reads（脏读），可以说是事务上的最低隔离级别：在普通的非锁定模式下 SELECT 的执行使我们看到的数据可能并不是查询发起时间点的数据，因而在这个隔离度下是非 Consistent Reads（一致性读）；

2. READ COMMITTED

这个事务隔离级别有些类似 Oracle 数据库默认的隔离级。属于语句级别的隔离，如通过 SELECT ... FOR UPDATE 和 SELECT ... LOCK IN SHARE MODE 来执行的请求仅仅锁定索引记录，而不锁定之前的间隙，因而允许在锁定的记录后自由地插入新记录。当然，这与 Innodb 的锁定实现机制有关。如果我们的 Query 可以很准确的通过索引定位到需要锁定的记录，则仅仅只需要锁定相关的索引记录，而不需要锁定该索引之前的间隙。但如果我们的 Query 通过索引检索的时候无法通过索引准确定位到需要锁定的记录，或者是一个基于范围的查询，InnoDB 就必须设置 next-key 或 gap locks 来阻塞其它用户对范围内的空隙插入。Consistent Reads 的实现机制与 Oracle 基本类似： 每一个 Consistent Read，甚至是同一个事务中的，均设置并作为它自己的最新快照。

这一隔离级别下，不会出现 Dirty Read，但是可能出现 Non-Repeatable Reads(不可重复读) 和 Phantom Reads（幻读）。

3. REPEATABLE READ

REPEATABLE READ 隔离级别是 InnoDB 默认的事务隔离级。SELECT ... FOR UPDATE, SELECT ... LOCK IN SHARE MODE, UPDATE, 和 DELETE ，这些以唯一条件搜索唯一索引的，只锁定所找到的索引记录，而不锁定该索引之前的间隙。 否则这些操作将使用 next-key 锁定，以 next-key 和 gap locks 锁定找到的索引范围，并阻塞其它用户的新建插入。在 Consistent Reads 中，与前一个隔离级相比这是一个重要的差别： 在这一级中，同一事务中所有的 Consistent Reads 均读取第一次读取时已确定的快照。这个约定就意味着如果在同一事务中发出几个无格式(plain)的 SELECTs ，这些 SELECT 的相互关系是一致的。

在 REPEATABLE READ 隔离级别下，不会出现 Dirty Reads，也不会出现 Non-Repeatable Reads，但是仍然存在 Phantom Reads 的可能性。

4. SERIALIZABLE

SERIALIZABLE 隔离级别是标准事务隔离级别中的最高级别。设置为 SERIALIZABLE 隔离级别之后，在事务中的任何时候所看到的数据都是事务启动时刻的状态，不论在这期间有没有其他事务已经修改了某些数据并提交。所以，SERIALIZABLE 事务隔离级别下，Phantom Reads 也不会出现。

以上四种事务隔离级别实际上就是 ANSI/ISO SQL92 标准所定义的四种隔离级别，Innodb 全部都为我们实现了。对于高并发应用来说，为了尽可能保证数据的一致性，避免并发可能带来的数据不一致问题，自然是事务隔离级别越高越好。但是，对于 Innodb 来说，所使用的事务隔离级别越高，实现复杂度自然就会更高，所需要做的事情也会更多，整体性能也就会更差。

所以，我们需要分析自己应用系统的逻辑，选择可以接受的最低事务隔离级别。以在保证数据安全一致性的同时达到最高的性能。

虽然 Innodb 存储引擎默认的事务隔离级别是 REPEATABLE READ，但实际上在我们大部分的应用场景下，都只需要 READ COMMITED 的事务隔离级别就可以满足需求了。

事务与 IO 的关系及优化

我想大部分人都清楚，Innodb 存储引擎通过缓存技术，将常用数据和索引缓存到内存中，这样我们在读取数据或者索引的时候就可以尽量减少物理 IO 来提高性能。那我们修改数据的时候 Innodb 是如何处理的呢，是否修改数据的时候 Innodb 是不是象我们常用的应用系统中的缓存一样，更改缓存中的数据的同时，将更改同时应用到相应的数据持久化系统中？

可能很多人都会有上面的这个疑问。实际上，Innodb 在修改数据的时候同样也只是修改 Buffer Pool 中的数据，并不是在一个事务提交的时候就将 BufferPool 中被修改的数据同步到磁盘，而是通过另外一种支持事务的数据库系统常用的手段，将修改信息记录到相应的事务日志中。

为什么不是直接将 Buffer Pool 中被修改的数据直接同步到磁盘，还有记录一个事务日志呢，这样不是反而增加了整体 IO 量了么？是的，对于系统的整体 IO 量而言，确实是有所增加。但是，对于系统的整体性能却有很大的帮助。

这里我们需要理解关于磁盘读写的两个概念：连续读写和随机读写。简单来说，磁盘的顺序读写就是将数据顺序的写入连续的物理位置，而随即读写则相反，数据需要根据各自的特定位置被写入各个位置，也就是被写入了并不连续的物理位置。对于磁盘来说，写入连续的位置最大的好处就是磁头所做的寻址动作很少，而磁盘操作中最耗费时间的就是磁头的寻址。所以，在磁盘操作中，连续读写操作比随即读写操作的性能要好很多。

我们的应用所修改的 Buffer Pool 中的数据都很随机，每次所做的修改都是一个或者少数几个数据页，多次修改的数据页也很少会连续。如果我们每次修改之后都将 Buffer Pool 中的数据同步到磁盘，那么磁盘就只能一直忙于频繁的随即读写操作。而事务日志在创建之初就是申请的连续的物理空间，而且每次写入都是紧接着之前的日志数据顺序的往后写入，基本上都是一个顺序的写入过程。所以，日志的写入操作远比同步 Buffer Pool 中被修改的数据要更快。

当然，由于事务日志都是通过几个日志文件轮循反复写入，而且每个日志文件大小固定，即使再多的日志也会有旧日志被新产生的日志覆盖的时候。所以，Buffer Pool 中的数据还是不可避免的需要被刷新到磁盘上进行持久化，而且这个持久化的动作必须在旧日志被新日志覆盖之前完成。只不过，随着被更新的数据（Dirty Buffer）的增加，需要刷新的数据的连续性就越高，所需要做的随机读写也就越少，自然，IO 性能也就得到了提升。

而且事务日志本身也有 Buffer（log buffer），每次事务日志的写入并不是直接写入到文件，也都是暂时先写入到 log buffer 中，然后再在一定的事件触发下才会同步到文件。当然，为了尽可能的减少事务日志的丢失，我们可以通过 innodb_log_buffer_size 参数来控制 log buffer 的大小。关于事务日志何时同步的说明稍后会做详细分析。

事务日志文件的大小与 Innodb 的整体 IO 性能有非常大的关系。理论上来讲，日志文件越大，则 Buffer Pool 所需要做的刷新动作也就越少，性能也越高。但是，我们也不能忽略另外一个事情，那就是当系统 Crash 之后的恢复。

事务日志的作用主要有两个，一个就是上面所提到的提高系统整体 IO 性能，另外一个就是当系统 Crash 之后的恢复。下面我们就来简单的分析一下当系统 Crash 之后，Innodb 是如何利用事务日志来进行数据恢复的。

Innodb 中记录了我们每一次对数据库中的数据及索引所做的修改，以及与修改相关的事务信息。同时还记录了系统每次 checkpoint 与 log sequence number（日志序列号）。

假设在某一时刻，我们的 MySQL Crash 了，那么很显然，所有 Buffer Pool 中的数据都会丢失，也包括已经修改且没有来得及刷新到数据文件中的数据。难道我们就让这些数据丢失么？当然不会，当 MySQL 从 Crash 之后再次启动，Innodb 会通过比较事务日志中所记录的 checkpoint 信息和各个数据文件中的 checkpoint 信息，找到最后一次 checkpoint 所对应的 log sequence number，然后通过事务日志中所记录的变更记录，将从 Crash 之前最后一次 checkpoint 往后的所有变更重新应用一次，同步所有的数据文件到一致状态，这样就找回了因为系统 Crash 而造成的所有数据丢失。当然，对于 log buffer 中未来得及同步到日志文件的变更数据就无法再找回了。系统 Crash 的时间离最后一次 checkpoint 的时间越长，所需要的恢复时间也就越长。而日志文件越大，Innodb 所做的 checkpoint 频率也越低，自然遇到长时间恢复的可能性也就越大了。

总的来说，Innodb 的事务日志文件设置的越大，系统的 IO 性能也就越高，但是当遇到 MySQL ，OS 或者主机 Crash 的时候系统所需要的恢复时间也就越长；反之，日志越小，IO 性能自然也就相对会差一些，但是当 MySQL，OS 或者主机 Crash 之后所需要的恢复时间也越小。所以，到底该将事务日志设置多大其实是一个整体权衡的问题，既要考虑到系统整体的性能，又要兼顾到 Crash 之后的恢复时间。一般来说，在我个人维护的环境中，比较偏向于将事务日志设置为 3 组，每个日志设置为 256MB 大小，整体效果还算不错。

前面所描述的场景还只是 MySQL Crash 的场景，我们所丢失的仅仅只是 Buffer Pool 中的数据。实际上 Innodb 事务日志也不一定每次事务提交或者回滚都保证会同步 log buffer 中的数据到文件系统并通知文件系统做文件同步操作。所以当我们的 OS Crash，或者是主机断点之后，事务日志写入文件系统 Buffer 中的数据还是可能会丢失，这种情况下，如果我们的事务日志没有及时同步文件系统刷新缓存中的数据到磁盘文件的话，就可能会产生日志数据丢失而造成数据永久性丢失的情况。

其实 Innodb 也早就考虑到了这种情况的存在，所以在系统中为我们设计了下面这个控制 Innodb 事务日志刷新方式的参数：innodb_flush_log_at_trx_commit。这个参数的主要功能就是让我们告诉系统，在什么情况下该通知文件系统刷新缓存中的数据到磁盘文件，可设置为如下三种值

- innodb_flush_log_at_trx_commit = 0，Innodb 中的 Log Thread 没隔 1 秒钟会将 log buffer 中的数据写入到文件，同时还会通知文件系统进行文件同步的 flush 操作，保证数据确实已经写入到磁盘上面的物理文件。但是，每次事务的结束（commit 或者是 rollback）并不会触发 Log Thread 将 log buffer 中的数据写入文件。所以，当设置为 0 的时候，当 MySQL Crash 和 OS Crash 或者主机断电之后，最极端的情况是丢失 1 秒时间的数据变更。
- innodb_flush_log_at_trx_commit = 1，这也是 Innodb 的默认设置。我们每次事务的结束都会触发 Log Thread 将 log buffer 中的数据写入文件并通知文件系统同步文件。这个设置是最安全的设置，能够保证不论是 MySQL Crash 还是 OS Crash 或者是主机断电都不会丢失任何已经提交的数据。
- innodb_flush_log_at_trx_commit = 2，当我们设置为 2 的时候，Log Thread 会在我们每次事务结束的时候将数据写入事务日志，但是这里的写入仅仅是调用了文件系统的文件写入操作。而我们的文件系统都是有缓存机制的，所以 Log Thread 的这个写入并不能保证内容真的已经写入到物理磁盘上面完成持久化的动作。文件系统什么时候会将缓存中的这个数据同步到物理磁

盘文件 Log Thread 就完全不知道了。所以，当设置为 2 的时候，MySQL Crash 并不会造成数据的丢失，但是 OS Crash 或者是主机断电后可能丢失的数据量就完全控制在文件系统上了。各种文件系统对于自己缓存的刷新机制各不相同，各位读者朋友如果有兴趣可以自行参阅相关的手册。

从上面的分析我们可以看出，当 innodb_flush_log_at_trx_commit 设置为 1 的时候是最安全的，但是由于所做的 IO 同步操作也最多，所以性能也是三种设置中最差的一种。如果设置为 0，则每秒有一次同步，性能相对高一些。如果设置为 2，可能性能是三这种最好的。但是也可能是出现鼓掌后丢失数据最多的。到底该如何设置设置，就要根据具体的场景来分析了。一般来说，如果完全不能接受数据的丢失，那么我们肯定会通过牺牲一定的性能来换取数据的安全性，选择设置为 1。而如果我们可以丢失很少量的数据(比如说 1 秒之内)，那么我们可以设置为 0。当然，如果大家觉得我们的 OS 足够稳定，主机硬件设备，而且主机的供电系统也足够安全，我们也可以将 innodb_flush_log_at_trx_commit 设置为 2 让系统的整体性能尽可能的高。

前面我们还提到了设置 Log Buffer 大小的参数 innodb_log_buffer_size。这里我们也简单的介绍一下 Log Buffer 的设置要领。Log Buffer 所存放的数据就是事务日志在写入文件之前在内存中的一个缓冲区域。所以理论上来讲，Log Buffer 越大，系统的性能也会越高。但是，由于触发 Log Thread 将 Log Buffer 中的数据写入文件的事件并不仅仅是 Log Buffer 空间用完的情况，还与 innodb_flush_log_at_trx_commit参数的设置有关。如果该参数设置为1或者2，那么我们的Log Buffer 中仅仅只需要保存单个事务的变更量与系统最高并发事务的乘积。也就是说，如果我们的系统同时进行修改的并发事务最高为 20 的话，那么我们的 Log Buffer 就只需要存放 20 个事务所作的变更。当然，如果我们设置为 0 的话，Log Buffer 中所需要存放的数据则是 1 秒内所有的变更量。所以，大家需要根据自己系统的具体环境来针对性分析 innodb_log_buffer_size 的设置大小。一般来说，如果不是特别高的事务并发度或者系统中都是大事务的话，8MB 的内存空间已经完全够用了。

11.2.3 数据存储优化

从“MySQL 存储引擎简介”一章中我们已经对 Innodb 存储引擎的物理结构有了一定的了解，这一节我们将通过分析 Innodb 的物理文件结构寻找可以优化的线索。

理解 Innodb 数据及索引文件存储格式

Innodb 存储引擎的数据（包括索引）存放在相同的文件中，这一点和 MySQL 默认存储引擎 MyISAM 的区别较大，后者分别存放于独立的文件。除此之外，Innodb 的数据存放格式也比较独特，每个 Innodb 表都会将主键以聚簇索引的形式创建。所有的数据都是以主键来作为升序排列在物理磁盘上面，所以主键查询并且以主键排序的查询效率也会非常高。

由于主键是聚族索引的缘故，Innodb 的基于主键的查询效率非常高。如果我们在创建一个 Innodb 存储引擎的表的时候并没有创建主键，那么 Innodb 会尝试在创建于我们表上面的其他索引，如果存在由单个 not null 属性列的唯一索引，Innodb 则会选择该索引作为聚族索引。如果也没有任何单个 not null 属性列的唯一索引，Innodb 会自动生成一个隐藏的内部列，该列会在每行数据上占用 6 个字节的存储长度。所以，实质上每个 Innodb 表都之少会有一个索引存在。

在 Innodb 上面出了聚族索引之外的索引被成为 secondary index，每个 secondary index 上都会包含有聚族索引的索引键信息，方便通过其他索引查找数据的时候能够更快的定位数据位置所在。

当然，聚族索引也并不是只有好处没有任何问题，要不然其他所有数据库早就大力推广了。聚族索引的最大问题就是当索引键被更新的时候，所带来的成本并不仅仅只是索引数据可能会需要移动，而是相关的所有记录的数据都需要移动。所以，为了性能考虑，我们应该尽可能不要更新 Innodb 的主键值。

Page

Innodb 存储引擎中的所有数据，不论是表还是索引，亦或是存储引擎自己的各种结构，都是以 page 作为最小物理单位来存放，每个 page 默认大小为 16KB。

extent

extent 是一个由多个连续的 page 组成一个物理存储单位。一般来说，每个 extent 为 64 个 page。

segment

segment 在 Innodb 存储引擎中实际上也代表“files”的意思，每个 segment 由一个或多个 extent 组成，而且每个 segment 都存放同一种数据。一般来说，每个表数据会存放于一个单独的 segment 中，实际上也就是每个聚族索引会存放于一个单独的 segment 中。

tablespace

tablespace 是 Innodb 中最大物理结构单位了，由多个 segment 组成。

当 tablespace 中的某个 segment 需要增长的时候，Innodb 最初仅仅分配某一个 extent 的前 32 个 pages，然后如果继续增长才会分配整个 extent 来使用。我们还可以通过执行如下命令来查看 Innodb 表空间的使用情况：

```
sky@localhost : example 01:26:43> SHOW TABLE STATUS  like 'test'\G
*************************** 1. row ***************************
           Name: test
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 8389019
 Avg_row_length: 29
    Data_length: 249298944
Max_data_length: 0
   Index_length: 123387904
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2008-11-15 01:26:43
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
```

```
Create_options:
        Comment: InnoDB free: 5120 kB
```

通过上面的显示，我们可以看出

虽然每个索引页（index page）大小为 16KB，但是实际上 Innodb 在第一次使用该 page 的时候，如果是一个顺序的索引插入，都会预留 1KB 的空间。而如果是随机插入的话，那么大约会使用 (8- 15/16) KB 的空间，而如果一个 Index page 在进行多次删除之后如果所占用的空间已经低于 8KB（1/2）的话，Innodb 会通过一定的收缩机制收缩索引，并释放该 index page。此外，每个索引记录中都存放了一个 6 字节的头信息，主要用于行锁定时候的记录以及各个索引记录的关联信息。

Innodb 在存放数据页的时候不仅仅只是存放我们实际定义的列，同时还会增加两个内部隐藏列，其中一个隐含列的信息主要为事务相关信息，会占用 6 个字节的长度。另外一个则占用 7 字节长度，主要用来存放一个指向 Undo Log 中的 Undo Segment 的指针相关信息，主要用于事务回滚，以及通过 Undo Segment 中的信息构造多版本数据页。

通过上面的信息，我们至少可以得出以下几点对性能有较大影响的地方：

1. 为了尽量减小 secondary index 的大小，提高访问效率，作为主键的字段所占用的存储空间越小越好，最好是 INTEGER 类型。当然这并不是绝对的，字符串类型的数据同样也可以作为 Innodb 表的主键；
2. 创建表的时候尽量自己指定相应的主键，让数据按照自己预设的顺序排序存放，一提高特定条件下的访问效率；
3. 尽可能不要在主键上面进行更新操作，减少因为主键值的变化带来数据的移动。
4. 尽可能提供主键条件进行查询；

分散 IO 提升磁盘响应

由于 Innodb 和其他非事务存储引擎相比在记录数据文件的同时还记录有相应的事务日志（Transaction Log），相当于增加的整体的 IO 量，虽然事务日志是以完全顺序的方式写入磁盘，但总是会有一定的 IO 消耗，所以对于没有做Raid 的磁盘系统来说，建议将数据文件和事务日志文件分别存放于不同的物理磁盘上面以降低磁盘的相互争用，提高整体 IO 性能。我们可以通过 innodb_log_group_home_dir 参数来指定 Innodb 日志存放位置，同时再通过设置数据文件位置 innodb_data_home_dir 参数来告诉 Innodb 我们希望将数据文件存放在哪里。

当然，如果我们使用独享表空间的话，Innodb 会为每个 Innodb 表创建一个表空间，并且会将该表空间存放在和“.frm”文件相同的路径下。不过幸运的是，Innodb 允许通过软链接的方式来访问数据或者日志文件。所以，如果我们有必要，甚至可以将每个表存放于单独的物理磁盘，然后再通过软链接的方式来告诉 Innodb 我们的实际文件在哪里。

当我们使用共享表空间的时候，最后一个数据文件必须是可以自动扩展的，这样就会带来一个疑问，在每次扩展的时候，到底该扩展多大空间性能会比较好呢？Innodb 给我们设计了 innodb_autoextend_increment 这个参数，让我们可以自行控制表空间文件每次增加的大小。

11.2.4 Innodb 其他优化

除了上面这些可以优化的地方之外，实际上 Innodb 还有其他一些可能影响到性能的参数设置：

- Innodb_flush_method
  用来设置 Innodb 打开和同步数据文件以及日志文件的方式，不过只有在 Linux & Unix 系统上面有效。系统默认值为 fdatasync，即 Innodb 默认通过 fsync()来 flush 数据和日志文件数据。
  此外，还可以设置为 O_DSYNC 和 O_DIRECT，当我们设置为 O_DSYNC，则系统以 O_SYNC 方式打开和刷新日志文件，通过 fsync()来打开和刷新数据文件。而设置为 O_DIRECT 的时候，则通过 O_DIRECT（Solaris 上为 directio()）打开数据文件，同时以 fsync()来刷新数据和日志文件。
  总的来说，innodb_flush_method 的不同设置主要影响的是 Innodb 在不同运行平台下进行 IO 操作的时候所调用的操作系统 IO 借口的区别。而不同的 IO 操作接口对数据的处理方式会有一定的区别，所以处理性能也会有一定的差异。一般来说，如果我们的磁盘是通过 RAID 卡做了硬件级别的 RAID，建议可以使用 O_DIRECT，可以一定程度上提高 IO 性能，但如果 RAID Cache 不够的话，还是需要谨慎对待。此外，根据 MySQL 官方手册上面的介绍，如果我们的存储环境是 SAN 环境，使用 O_DIRECT 有可能会反而使性能降低。对于支持 O_DSYNC 的平台，也可以尝试设置为 O_DSYNC 方式看是否能对写 IO 性能有所帮助。

- innodb_thread_concurrency
  这个参数主要控制 Innodb 内部的并发处理线程数量的最大值，系统内部会有相应的检测机制进行检测控制并发线程数量，Innodb 建议设置为 CPU 个数与磁盘个数之和。但是这个参数一直是一个非常有争议的参数，而且还有一个非常著名的 BUG（#15815）一直被认为就于 innodb_thread_concurrency 参数所控制的内容相关。从该参数在系统中的默认值的变化我们也可以看出即使是 Innodb 开发人员也并不是很清楚到底该将 innodb_thread_concurrency 设置为多少合适。在 MySQL5.0.8 之前，默认值为 8，从 MySQL5.0.8 开始到 MySQL5.0.18，默认值又被更改为 20，然后在 MySQL5.0.19 和 MySQL5.0.20 两个版本中又默认设置为 0。之后，从 MySQL5.0.21 开始默认值再次被更改回 8。
  innodb_thread_concurrency 参数的设置范围是 0～1000，但是在 MySQL5.0.19 之前的版本，只要该值超过 20，Innodb 就会认为不需要对并发线程数做任何限制，也就是说 Innodb 不会再进行并行线程的数目检查。同样，我们也可以通过设置为 0 来禁用并行线程检查，完全让 Innodb 自己根据实际需要创建并行线程，而且在不少场景下设置为 0 还是一个非常不错的选择，尤其是当系统写 IO 压力较大的时候。
  总的来说，innodb_thread_concurrency 参数的设置并没有一个很好的规则来判断什么场景该设置多大，完全需要通过不断的调整尝试，寻找出适合自己应用的设置。

- autocommit
  autocommit 的用途我想大家应该都很清楚，就是当我们将该参数设置为 true(1)之后，在我们每次执行完一条会修改数据的 Query 之后，系统内部都会自动提交该操作，基本上可以理解为屏蔽了事务的概念。
  设置 aotucommit 为 true（1）之后，我们的提交相对于自己手工控制 commit 时机来说可能会变得要频繁很多。这样带来的直接影响就是 Innodb 的事务日志可能会需要非常频繁的执行磁盘同

步操作，当然还与 innodb_flush_log_at_trx_commit 参数的设置相关。
一般来说，在我们通过 LOAD ... INFILE ... 或者其他的某种方式向 Innodb 存储引擎的表加载数据的时候，将 autocommit 设置为 false 可以极大的提高加载性能。而在正常的应用中，也最好尽量通过自行控制事务的提交避免过于频繁的日志刷新来保证性能。

11.2.5 Innodb 性能监控

我们可以通过执行“SHOW INNODB STATUS”命令来获取比较详细的系统当前 Innodb 性能状态，如下：

```
    sky@localhost : example 03:11:19> show innodb status\G
*************************** 1. row ***************************
Status:
=====================================
081115 15:56:30 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 10 seconds
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 720, signal count 719
Mutex spin waits 0, rounds 16962, OS waits 460
RW-shared spins 489, OS waits 244; RW-excl spins 3, OS waits 3
------------
TRANSACTIONS
------------
Trx id counter 0 11605
Purge done for trx's n:o < 0 11604 undo n:o < 0 0
History list length 10
Total number of lock structs in row lock hash table 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, process no 13383, OS thread id 2892274576
MySQL thread id 9, query id 54 localhost sky
show innodb status
--------
FILE I/O
--------
I/O thread 0 state: waiting for i/o request (insert buffer thread)
I/O thread 1 state: waiting for i/o request (log thread)
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (write thread)
Pending normal aio reads: 0, aio writes: 0,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
```

```
Pending flushes (fsync) log: 0; buffer pool: 0
1123 OS file reads, 2791 OS file writes, 1941 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2,
0 inserts, 0 merged recs, 0 merges
Hash table size 138401, used cells 2, node heap has 1 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number 0 1072999334
Log flushed up to   0 1072999334
Last checkpoint at  0 1072999334
0 pending log writes, 0 pending chkp writes
1301 log i/o's done, 0.00 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 58787017; in additional pool allocated 1423616
Buffer pool size   2048
Free buffers       803
Database pages     1244
Modified db pages  0
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 15923, created 22692, written 23332
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
1 read views open inside InnoDB
Main thread process no. 13383, id 2966408080, state: waiting for server activity
Number of rows inserted 8388614, updated 0, deleted 0, read 8388608
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

通过上面的输出，我们可以看到整个信息分为 7 个部分，分别说明如下：

- SEMAPHORES，这部分主要显示系统中当前的信号等待信息以及各种等待信号的统计信息，这部分输出的信息对于我们调整 innodb_thread_concurrency 参数有非常大的帮助，当等待信号量非常大的时候，可能就需要禁用并发线程检测设置 innodb_thread_concurrency=0;
- TRANSACTIONS，这里主要展示系统的锁等待信息和当前活动事务信息。通过这部分输出，我们可以查追踪到死锁的详细信息;
- FILE I/O，文件 IO 相关的信息，主要是 IO 等待信息;
- INSERT BUFFER AND ADAPTIVE HASH INDEX；显示插入缓存当前状态信息以及自适应Hash Index 的状态;
- LOG，Innodb 事务日志相关信息，包括当前的日志序列号（Log Sequence Number），已经刷新同步到哪个序列号，最近的 Check Point 到哪个序列号了。除此之外，还显示了系统从启动到现在已经做了多少次 Ckeck Point，多少次日志刷新;
- BUFFER POOL AND MEMORY，这部分主要显示 Innodb Buffer Pool 相关的各种统计信息，以及其他一些内存使用的信息;
- ROW OPERATIONS，顾名思义，主要显示的是与客户端的请求 Query 和这些 Query 所影响的记录统计信息。

这里只是对输出做了一个简单的介绍，如果各位读者朋友希望更深入的了解相应的细节，建议查阅 Innodb 相关手册，此外，《High Performance MySQL》作者之一 Peter Zaitsev 有一篇叫做“SHOW INNODB STATUS walk through”的文件专门做了较为详细的分析，大家可以通过访问 http://www.mysqlperformanceblog.com/网址去了解。

当然，如果我们总是要通过不断执行“SHOW INNODB STATUS”命令来获取这样的性能信息是在是有些麻烦，所以 Innodb 存储引擎为我们设计了一个比较奇怪的方式来持续获取该信息并输出到 MySQL Error Log 中。

实现方式就是通过创建一个名为 innodb_monitor，存储引擎为 Innodb 的表，够奇特吧，如下：

```
CREATE TABLE innodb_monitor(a int) ENGINE=INNODB;
```

当我们创建这样一个表之后，Innodb 就会每过 15 秒输出一次 Innodb 整体状态信息，也就是上面所展示的信息到 Error Log 中。我们可以通过删除该表停止该 Monitor 功能，如下：

```
DROP TABLE innodb_monitor;
```

除此之外，我们还可以通过相同的方式打开和关闭 innodb_tablespace_monitor,innodb_lock_monitor,innodb_table_monitor这三种监控功能，各位读者朋友可以自行尝试。

通过上面的各种监控信息的输出信息，我们可以比较详细的了解到 Innodb 当前的运行状态，帮助我们及时发现性能问题。

11.3 小结

MyISAM 和 Innodb 两种存储引擎各有特点，很多使用者对这两种存储引擎各有偏好，认为某一种要优于另外一种，实际上这是比较片面的认识。两种存储引擎各自都存在对方没有的优点，也存在自身的缺点，我们只有充分了解了各自的优缺点之后，在实际应用环境中根据不同的需要选择不同的存储引擎，才能将 MySQL 用到最好。

此外，随着 MySQL Cluster 的不断成熟，除了上面详细分析的两种存储引擎之外，实际上还有 NDB Cluster 存储引擎正在被越来越多的使用，关于 NDB Cluster 相关的内容，将在架构设计篇中再进行比较详细的介绍。

第 12 章 MySQL 可扩展设计的基本原则

前言：

随着信息量的飞速增加，硬件设备的发展已经慢慢的无法跟上应用系统对处理能力的要求了。此时，我们如何来解决系统对性能的要求？只有一个办法，那就是通过改造系统的架构体系，提升系统的扩展能力，通过组合多个低处理能力的硬件设备来达到一个高处理能力的系统，也就是说，我们必须进行可扩展设计。可扩展设计是一个非常复杂的系统工程，所涉及的各个方面非常的广泛，技术也较为复杂，可能还会带来很多其他方面的问题。但不管我们如何设计，不管遇到哪些问题，有些原则我们还是必须确保的。本章就将可扩展设计过程中需要确保的原则做一个简单的介绍。

12.1 什么是可扩展性

在讨论可扩展性之前，可能很多朋有会问：常听人说起某某网站某某系统在可扩展性方面设计的如何如何好，架构如何如何出色，到底什么是扩展？怎样算是可扩展？什么又是可扩展性呢？其实也就是大家常听到的 Scale，Scalable 和 Scalability 这三个词。

从数据库的角度来说，Scale（扩展）就是让我们的数据库能够提供更强的服务能力，更强的处理能力。而 Scalable（可扩展）则是表明数据库系统在通过相应升级（包括增加单机处理能力或者增加服务器数量）之后能够达到提供更强处理能力。在理论能上来说，任何数据库系统都是 Scalable 的，只不过是所需要的实现方式不一样而已。最后，Scalability（扩展性）则是指一个数据库系统通过相应的升级之后所带来处理能力提升的难以程度。虽然理论上任何系统都可以通过相应的升级来达到处理能力的提升，但是不同的系统提升相同的处理能力所需要的升级成本（资金和人力）是不一样的，这也就是我们所说的各个数据库应用系统的 Scalability 存在很大的差异。

在这里，我所说的不同数据库应用系统并不是指数据库软件本身的不同（虽然数据库软件不同也会存在 Scalability 的差异），而是指相同数据库软件的不同应用架构设计，这也正是本章以及后面几张将会所重点分析的内容。

首先，我们需要清楚一个数据库据系统的扩展性实际上是主要体现在两个方面，一个是横向扩展，另一个则是纵向扩展，也就是我们常说的 Scale Out 和 Scale Up。

Scale Out 就是指横向的扩展，向外扩展，也就是通过增加处理节点的方式来提高整体处理能力，说的更实际一点就是通过增加机器来增加整体的处理能力。

Scale Up 则是指纵向的扩展，向上扩展，也就是通过增加当前处理节点的处理能力来提高整体的处理能力，说白了就是通过升级现有服务器的配置，如增加内存，增加 CPU，增

加存储系统的硬件配置，或者是直接更换为处理能力更强的服务器和更为高端的存储系统。

通过比较两种 Scale 方式，我们很容易看出各自的优缺点。

- Scale Out 优点：
  1. 成本低，很容易通过价格低廉的 PC Server 搭建出一个处理能力非常强大的计算集群；
  2. 不太容易遇到瓶颈，因为很容易通过添加主机来增加处理能力；
  3. 单个节点故障对系统整体影响较小；也存在缺点，更多的计算节点，大部分时候都是服务器主机，这自然会带来整个系统维护复杂性的提高，在某些方面肯定会增加维护成本，而且对应用系统的架构要求也会比 Scale Up 更高，需要集群管理软件的配合。
- Scale Out 缺点：
  1. 处理节点多，造成系统架构整体复杂度提高，应用程序复杂度提高；
  2. 集群维护难以程度更高，维护成本更大；
- Scale Up 优点：
  1. 处理节点少，维护相对简单；
  2. 所有数据都集中在一起，应用系统架构简单，开发相对容易；
- Scale Up 缺点
  1. 高端设备成本高，且竞争少，容易受到厂家限制；
  2. 受到硬件设备发展速度限制，单台主机的处理能力总是有极限的，容易遇到最终无法解决的性能瓶颈；
  3. 设备和数据集中，发生故障后的影响较大；

从短期来看，Scale Up 会有更大的优势，因为可以简化运维成本，简化系统架构和应用系统的开发，对技术方面的要求要会更简单一些。

但是，从长远影响来看，Scale Out 会有更大的优势，而且也是系统达到一个规模之后的必然趋势。因为不管怎样，单台机器的处理能力总是会受到硬件技术的限制，而硬件技术的发展速度总是有限的，很多时候很难跟得上业务发展的速度。而且越是高处理能力的高端设备，其性价比总是会越差。所以通过多台廉价的 PC Server 构建高处理能力的分布式集群，总是会成为各个公司节约成本，提高整体处理能力的一个目标。虽然在实现这个目标的时候可能会遇到各种各样的技术问题，但总是值得去研究实践的。

后面的内容，我们将重点针对 Scale Out 方面来进行分析设计。要能够很好的 Scale Out，势必需要进行分布式的系统设计。对于数据库，要想较好的 Scale Out，我们只有两个方向，一个是通过数据的不断复制来实现很多个完全一样的数据源来进行扩展，另一个就是通过将一个集中的数据源切分成很多个数据源来实现扩展。

下面我们先看看在设计一个具有很好的 Scalability 的数据库应用系统架构方面，需要遵循一些什么样的原则。

12.2 事务相关性最小化原则

搭建分布式数据库集群的时候，很多人都会比较关心事务的问题。毕竟事务是数据库中非常核心的一个功能。

在传统的集中式数据库架构中，事务的问题非常好解决，可以完全依赖数据库本身非常成熟的事务机制来保证。但是一旦我们的数据库作为分布式的架构之后，很多原来在单一数据库中所完成的事务现在可能需要跨多个数据库主机，这样原来单机事务可能就需要引入分布式事务的概念。

但是大家肯定也有一些了解，分布式事务本身就是一个非常复杂的机制，不管是商业的大型数据库系统还是各开源数据库系统，虽然大多数数据库厂家基本上都实现了这个功能，但或多或少都存在各种各样的限制。而且也存在一些 Bug，可能造成某些事务并不能很好的保证，或者是不能顺利的完成。

这时候，我们可能就需要寻求其他的替代方案来解决这个问题，毕竟事务是不可忽视的，不关我们如何去实现，总是需要实现的。

就目前来说，主要存在的一些解决方案主要有以下三种：

第一、进行 Scale Out 设计的时候合理设计切分规则，尽可能保证事务所需数据在同一个 MySQL Server 上，避免分布式事务。

如果可以在设计数据切分规则的时候就做到所有事务都能够在单个 MySQL Server 上面完成，我们的业务需求就可以比较容易的实现，应用程序就可以做到通过最少的调整来满足架构的改动，使整体成本大大减少。毕竟，数据库架构改造并不仅仅只是 DBA 的事情，还需要很多外围的配合与支持。即使是在设计一个全新系统的时候，我们同样要考虑到各个环境各项工作的整体投入，既要考虑数据库本身的成本投入，同时也要考虑到相应的开发代价。如果各环节之间出现“利益”冲突，那我们就必须要作出一个基于后续扩展以及总体成本的权衡，寻找出一个最适合当前阶段平衡点。

不过，即使我们的切分规则设计的再高明，也很难让所有的事务所需的数据都在同一个 MySQL Server 上。所以，虽然这种解决方案所需要付出的成本最小，但大多数时候也只能兼顾到一些大部分的核心事务，也不是一个很完美的解决方案。

第二、大事务切分成多个小事务，数据库保证各个小事务的完整性，应用控制各个小事务之间的整体事务完整性。

和上一个方案相比，这个方案所带来的应用改造就会更多，对应用的要求也会更为苛刻。应用不仅需要分拆原来的很多大事务，同时还需要保证各个小事务的之间的完整性。也就是说，应用程序自己需要具有一定的事务能力，这无疑会增加应用程序的技术难度。

但是，这个方案也有不少自己的优势。首先我们的数据的切分规则就会更为简单，很难遇到限制。而且更简单，就意味着维护成本更低。其次，没有数据切分规则的太多限制，数据库方面的可扩展性也会更高，不会受到太多的约束，当出现性能瓶颈的时候可以快速进行进一步拆分现有数据库。最后，数据库做到离实际业务逻辑更远，对后续架构扩展也就更为有利。

第三、结合上述两种解决方案，整合各自的优势，避免各自的弊端。

前面两种解决方案都存在各自的优缺点，而且基本上都是相互对立的，我们完全可以利用两者各自的优势，调整两个方案的设计原则，在整个架构设计中做一个平衡。比如我们可以在保证部分核心事务所需数据在同一个 MySQL Server 上，而其他并不是特别重要的事务，则通过分拆成小事务和应用系统结合来保证。而且，对于有些并不是特别重要的事务，我们也可以通过深入分析，看是否不可避免一定需要使用事务。

通过这样相互平衡设计的原则，我们既可以避免应用程序需要处理太多的小事务来保证其整体的完整性，同时也能够避免拆分规则太多复杂而带来后期维护难度的增加及扩展性受阻的情况。

当然，并不是所有的应用场景都非要结合以上两种方案来解决。比如对于那些对事务要求并不是特别严格，或者事务本身就非常简单的应用，就完全可以通过稍加设计的拆分规则就可满足相关要求，我们完全可以仅仅使用第一中方案，就可以避免还需要应用程序来维护某些小事务的整体完整性的支持。这在很大程度上面可以降低应用程序的复杂度。

而对于那些事务关系非常复杂，数据之间的关联度非常高的应用，我们也就没有必要为了保持事务数据能够集中而努力设计，因为不管我们如何努力，都很难满足要求，大都是遇到顾此失彼的情景。对于这种情况，我们还不如让数据库方面尽可能保持简洁，而让应用程序做出一些牺牲。

在当前很多大型的互联网应用中，不论是上面哪一种解决方案的使用案例都有，如大家所熟知的 Ebay，在很大程度上就是第三种结合的方案。在结合过程中以第二种方案为主，第一种方案为辅。选择这样的架构，除了他们应用场景的需求之外，其较强的技术实力也为开发足够强壮的应用系统提供了保证。又如某国内大型的 BBS 应用系统（不便公开其真实名称），其事务关联性并不是特别的复杂，各个功能模块之间的数据关联性并不是特别的高，就是完全采用第一种解决方案，完全通过合理设计数据拆分的规则来避免事务的数据源跨多个 MySQL Server。

最后，我们还需要明白一个观点，那就是事务并不是越多越好，而是越少越好越小越好。不论我们使用何种解决方案，那就是在我们设计应用程序的时候，都需要尽可能做到让数据的事务相关性更小，甚至是不需要事务相关性。当然，这只是相对的，也肯定只有部分数据能够做到。但可能就是某部分数据做到了无事务相关性之后，系统整体复杂度就会降低很大一个层次，应用程序和数据库系统两方面都可能少付出很多的代价。

12.3 数据一致性原则

不论是 Scale Up 还是 Scale Out，不论我们如何设计自己的架构，保证数据的最终一致性都是绝对不能违背的原则，保证这个原则的重要性我想各位读者肯定也都是非常明白清楚的。

而且，数据一致性的保证就像事务完整性一样，在我们对系统进行 Scale Out 设计的时候，也可能会遇到一些问题。当然，如果是 Scale Up，可能就很少会遇到这类麻烦了。当然，在很多人眼中，数据的一致性在某种程度上面也是属于事务完整性的范畴。不过这里为了突出其重要性和相关特性，我还是将他单独提出来分析。

那我们又如何在 Scale Out 的同时又较好的保证数据一致性呢？很多时候这个问题和保证事务完整性一样让我们头疼，也同样受到了很多架构师的关注。经过很多人的实践，大家最后总结出了 BASE 模型。即：基本可用，柔性状态，基本一致和最终一致。这几个词看着挺复杂挺深奥，其实大家可以简单的理解为非实时的一致性原则。

也就是说，应用系统通过相关的技术实现，让整个系统在满足用户使用的基础上，允许数据短时间内处于非实时状态，而通过后续技术来保证数据在最终保证处于一致状态。这个理论模型说起来确实听简单，但实际实现过程中我们也会遇到不少难题。

首先，第一个问题就是我们需要让所有数据都是非实时一致吗？我想大多数读者朋友肯定是投反对票的。那如果不是所有的数据都是非实时一致，那我们又该如何来确定哪些数据需要实时一致哪些数据又只需要非实时的最终一致呢？其实这基本可以说是一个各模块业务优先级的划分，对于优先级高的自然是规属于保证数据实时一致性的阵营，而优先级略低的应用，则可以考虑划分到允许短时间端内不一致而最终一致的阵营。这是一个非常棘手的问题。我们不能随便拍脑袋就决定，而是需要通过非常详细的分析和仔细的评估才能作出决定。因为不是所有数据都可以出现在系统能不短时间段内不一致状态，也不是所有数据都可以通过后期处理的使数据最终达到一致的状态，所以之少这两类数据就是需要实时一致的。而如何区分出这两类数据，就必须经过详细的分析业务场景商业需求后进行充分的评估才能得出结论。

其次，如何让系统中的不一致数据达到最终一致？一般来说，我们必须将这类数据所设计到的业务模块和需要实时一致数据的业务模块明确的划分开来。然后通过相关的异步机制技术，利用相应的后台进程，通过系统中的数据，日志等信息将当前并不一致的数据进行进一步处理，使最终数据处于完全一致状态。对于不同的模块，使用不同的后台进程，既可以避免数据出现紊乱，也可以并发执行，提高处理效率。如对用户的消息通知之类的信息，就没有必要做到严格的实时一致性，只需要现记录下需要处理的消息，然后让后台的处理进程依次处理，避免造成前台业务的拥塞。

最后，避免实时一致与最终一致两类数据的前台在线交互。由于两类数据状态的不一致性，很可能会导致两类数据在交互过程中出现紊乱，应该尽量让所有非实时一致的数据和实时一致数据在应用程序中得到有效的隔离。甚至在有些特别的场景下，记录在不同的 MySQL

Server 中来进行物理隔离都是有必要的。

12.4 高可用及数据安全原则

除了上面两个原则之外，我还想提一下系统高可用及数据安这两方面。经过我们的 Scale Out 设计之后，系统整体可扩展性确实是会得到很大的提高，整体性能自然也很容易得到较大的改善。但是，系统整体的可用性维护方面却是变得比以前更为困难。因为系统整体架构复杂了，不论是应用程序还是数据库环境方面都会比原来更为庞大，更为复杂。这样所带来的最直接影响就是维护难度更大，系统监控更难。

如果这样的设计改造所带来的结果是我们系统经常性的 Crash,经常性的出现 Down 机事故，我想大家肯定是无法接受的，所以我们必须通过各种技术手段来保证系统的可用性不会降低，甚至在整体上有所提高。

所以，这里很自然就引出了我们在进行 Scale Out 设计过程中另一个原则，也就是高可用性的原则。不论如何调整设计系统的架构，系统的整体可用性不能被降低。

其实在讨论系统可用性的同时，还会很自然的引出另外一个与之密切相关的原则，那就是数据安全原则。要想达到高可用，数据库中的数据就必须是足够安全的。这里所指的安全并不针对恶意攻击或者窃取方面来说，而是针对异常丢失。也就是说，我们必须保证在出现软/硬件故障的时候，能够保证我们的数据不会出现丢失。数据一旦丢失，根本就无可用性可言了。而且，数据本身就是数据库应用系统最核心的资源，绝对不能丢失这一原则也是毋庸置疑的。

要确保高可用及数据安全原则，最好的办法就是通过冗余机制来保证。所有软硬件设备都去除单点隐患，所有数据都存在多份拷贝。这样才能够较好的确保这一原则。在技术方面，我们可以通过 MySQL Replication，MySQL Cluster 等技术来实现。

12.5 小结

不论我们如何设计架构，不管我们的可扩展性如何变化，本章中所提到的一些原则都是非常重要的。不论是解决某些问题的原则，还是保证性的原则，不论是保证可用性的原则，还是保证数据安全的原则，我们都应该在设计中时时刻刻都关注，谨记。

MySQL 数据库之所以在互联网行业如此火爆，除了其开源的特性，使用简单之外，还有一个非常重要的因素就是在扩展性方面有较大的优势。其不同存储引擎各自所拥有的特性可以应对各种不同的应用场景。其 Replication 以及 Cluster 等特性更是提升扩展性非常有效的手段。

第 13 章 可扩展性设计之 MySQL Replication

前言：

MySQL Replication 是 MySQL 非常有特色的一个功能，他能够将一个 MySQL Server 的 Instance 中的数据完整的复制到另外一个 MySQL Server 的 Instance 中。虽然复制过程并不是实时而是异步进行的，但是由于其高效的性能设计，延时非常之少。MySQL 的 Replication 功能在实际应用场景中被非常广泛的用于保证系统数据的安全性和系统可扩展设计中。本章将专门针对如何利用 MySQL 的 Replication 功能来提高系统的扩展性进行详细的介绍。

13.1 Replication 对可扩展性设计的意义

在互联网应用系统中，扩展最为方便的可能要数最基本的 Web 应用服务了。因为 Web 应用服务大部分情况下都是无状态的，也很少需要保存太多的数据，当然 Session 这类信息比较例外。所以，对于基本的 Web 应用服务器很容易通过简单的添加服务器并复制应用程序来做到 Scale Out。

而数据库由于其特殊的性质，就不是那么容易做到方便的 Scale Out。当然，各个数据库厂商也一直在努力希望能够做到自己的数据库软件能够像常规的应用服务器一样做到方便的 Scale Out，也确实做出了一些功能，能够基本实现像 Web 应用服务器一样的 Scalability，如很多数据库所支持的逻辑复制功能。

MySQL 数据库也为此做出了非常大的努力,MySQL Replication 功能主要就是基于这一目的所产生的。通过 MySQL 的 Replication 功能，我们可以非常方便的将一个数据库中的数据复制到很多台 MySQL 主机上面，组成一个 MySQL 集群，然后通过这个 MySQL 集群来对外提供服务。这样，每台 MySQL 主机所需要承担的负载就会大大降低，整个 MySQL 集群的处理能力也很容易得到提升。

为什么通过 MySQL 的 Replication 可以做到 Scale Out 呢？主要是因为通过 MySQL 的 Replication,可以将一台 MySQL 中的数据完整的同时复制到多台主机上面的 MySQL 数据库中，并且正常情况下这种复制的延时并不是很长。当我们各台服务器上面都有同样的数据之后，应用访问就不再只能到一台数据库主机上面读取数据了，而是访问整个 MySQL 集群中的任何一台主机上面的数据库都可以得到相同的数据。此外还有一个非常重要的因素就是 MySQL 的复制非常容易实施，也非常容易维护。这一点对于实施一个简单的分布式数据库集群是非常重要的，毕竟一个系统实施之后的工作主要就是维护了，一个维护复杂的系统肯定不是一个受欢迎的系统。

13.2 Replication **机制的实现原理**

要想用好一个系统，理解其实现原理是非常重要的事情，只有理解了其实现原理，我们才能够扬长避短，合理的利用，才能够搭建出最适合我们自己应用环境的系统，才能够在系统实施之后更好的维护他。

下面我们分析一下 MySQL Replication 的实现原理。

13.2.1 Replication 线程

Mysql 的 Replication 是一个异步的复制过程，从一个 Mysql instace（我们称之为 Master）复制到另一个 Mysql instance（我们称之 Slave）。在 Master 与 Slave 之间的实现整个复制过程主要由三个线程来完成,其中两个线程(Sql 线程和 IO 线程)在 Slave 端，另外一个线程（IO 线程）在 Master 端。

要实现 MySQL 的 Replication ，首先必须打开 Master 端的 Binary Log（mysql-bin.xxxxxx）功能，否则无法实现。因为整个复制过程实际上就是 Slave 从 Master 端获取该日志然后再在自己身上完全顺序的执行日志中所记录的各种操作。打开 MySQL 的 Binary Log 可以通过在启动 MySQL Server 的过程中使用“—log-bin”参数选项,或者在 my.cnf 配置文件中的 mysqld 参数组（[mysqld]标识后的参数部分）增加 “log-bin” 参数项。

MySQL 复制的基本过程如下：

1. Slave 上面的 IO 线程连接上 Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；

2. Master 接收到来自 Slave 的 IO 线程的请求后，通过负责复制的 IO 线程根据请求信息读取指定日志指定位置之后的日志信息，返回给 Slave 端的 IO 线程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息在 Master 端的 Binary Log 文件的名称以及在 Binary Log 中的位置；

3. Slave 的 IO 线程接收到信息后，将接收到的日志内容依次写入到 Slave 端的 Relay Log 文件(mysql-relay-bin.xxxxxx)的最末端，并将读取到的 Master 端的 bin-log 的文件名和位置记录到 master-info 文件中，以便在下一次读取的时候能够清楚的高速 Master“我需要从某个 bin-log 的哪个位置开始往后的日志内容，请发给我”

4. Slave 的 SQL 线程检测到 Relay Log 中新增加了内容后，会马上解析该 Log 文件中的内容成为在 Master 端真实执行时候的那些可执行的 Query 语句，并在自身执行这些 Query。这样，实际上就是在 Master 端和 Slave 端执行了同样的 Query，所以两端的数据是完全一样的。

实际上，在老版本中，MySQL 的复制实现在 Slave 端并不是由 SQL 线程和 IO 线程这两个线程共同协作而完成的，而是由单独的一个线程来完成所有的工作。但是 MySQL 的工程师们很快发现，这样做存在很大的风险和性能问题，主要如下：

首先，如果通过一个单一的线程来独立实现这个工作的话，就使复制 Master 端的，Binary Log 日志，以及解析这些日志，然后再在自身执行的这个过程成为一个串行的过程，性能自然会受到较大的限制，这种架构下的 Replication 的延迟自然就比较长了。

其次，Slave 端的这个复制线程从 Master 端获取 Binary Log 过来之后，需要接着解析这些内容，还原成 Master 端所执行的原始 Query，然后在自身执行。在这个过程中，Master 端很可能又已经产生了大量的变化并生成了大量的 Binary Log 信息。如果在这个阶段 Master 端的存储系统出现了无法修复的故障，那么在这个阶段所产生的所有变更都将永远的丢失，无法再找回来。这种潜在风险在 Slave 端压力比较大的时候尤其突出，因为如果 Slave 压力比较大，解析日志以及应用这些日志所花费的时间自然就会更长一些，可能丢失的数据也就会更多。

所以，在后期的改造中，新版本的 MySQL 为了尽量减小这个风险，并提高复制的性能，将 Slave 端的复制改为两个线程来完成，也就是前面所提到的 SQL 线程和 IO 线程。最早提出这个改进方案的是 Yahoo!的一位工程师“Jeremy Zawodny”。通过这样的改造，这样既在很大程度上解决了性能问题，缩短了异步的延时时间，同时也减少了潜在的数据丢失量。

当然，即使是换成了现在这样两个线程来协作处理之后，同样也还是存在 Slave 数据延时以及数据丢失的可能性的，毕竟这个复制是异步的。只要数据的更改不是在一个事务中，这些问题都是存在的。

如果要完全避免这些问题，就只能用 MySQL 的 Cluster 来解决了。不过 MySQL 的 Cluster 知道笔者写这部分内容的时候，仍然还是一个内存数据库的解决方案，也就是需要将所有数据包括索引全部都 Load 到内存中，这样就对内存的要求就非常大的大，对于一般的大众化应用来说可实施性并不是太大。当然，在之前与 MySQL 的 CTO David 交流的时候得知，MySQL 现在正在不断改进其 Cluster 的实现，其中非常大的一个改动就是允许数据不用全部 Load 到内存中，而仅仅只是索引全部 Load 到内存中，我想信在完成该项改造之后的 MySQL Cluster 将会更加受人欢迎，可实施性也会更大。

13.2.2 复制实现级别

MySQL 的复制可以是基于一条语句（Statement Level），也可以是基于一条记录（Row level），可以在 MySQL 的配置参数中设定这个复制级别，不同复制级别的设置会影响到 Master 端的 Binary Log 记录成不同的形式。

1. Row Level：Binary Log 中会记录成每一行数据被修改的形式，然后在 Slave 端再对相同的数据进行修改。

优点：在 Row Level 模式下，Binary Log 中可以不记录执行的 sql 语句的上下文相关的信息，仅仅只需要记录那一条记录被修改了，修改成什么样了。所以 Row Level 的日志内容会非常清楚的记录下每一行数据修改的细节，非常容易理解。而且不会出现某些特定情况下的存储过程，或 function，以及 trigger 的调用和触发无法被正确复制的问题。

缺点：Row Level 下，所有的执行的语句当记录到 Binary Log 中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容，比如有这样一条 update 语句：UPDATE group_message SET group_id = 1 where group_id = 2，执行之后，日志中记录的不是这条 update 语句所对应的事件（MySQL 以事件的形式来记录 Binary Log 日志），而是这条语句所更新的每一条记录的变化情况，这样就记录成很多条记录被更新的很多个事件。自然，Binary Log 日志的量就会很大。尤其是当执行 ALTER TABLE 之类的语句的时候，产生的日志量是惊人的。因为 MySQL 对于 ALTER TABLE 之类的 DDL 变更语句的处理方式是重建整个表的所有数据，也就是说表中的每一条记录都需要变动，那么该表的每一条记录都会被记录到日志中。

2. Statement Level:每一条会修改数据的 Query 都会记录到 Master 的 Binary Log 中。Slave 在复制的时候 SQL 线程会解析成和原来 Master 端执行过的相同的 Query 来再次执行。

优点：Statement Level 下的优点首先就是解决了 Row Level 下的缺点，不需要记录每一行数据的变化，减少 Binary Log 日志量，节约了 IO 成本，提高了性能。因为他只需要记录在 Master 上所执行的语句的细节，以及执行语句时候的上下文的信息。

缺点：由于他是记录的执行语句，所以，为了让这些语句在slave端也能正确执行，那么他还必须记录每条语句在执行的时候的一些相关信息，也就是上下文信息，以保证所有语句在slave端杯执行的时候能够得到和在master端执行时候相同的结果。另外就是，由于Mysql现在发展比较快，很多的新功能不断的加入，使mysql得复制遇到了不小的挑战，自然复制的时候涉及到越复杂的内容，bug也就越容易出现。在statement level下，目前已经发现的就有不少情况会造成mysql的复制出现问题，主要是修改数据的时候使用了某些特定的函数或者功能的时候会出现，比如：sleep()函数在有些版本中就不能真确复制，在存储过程中使用了last_insert_id()函数，可能会使slave和master上得到不一致的id等等。由于row level是基于每一行来记录的变化，所以不会出现类似的问题。

从官方文档中看到，之前的 MySQL 一直都只有基于 Statement 的复制模式，直到5.1.5版本的 MySQL 才开始支持Row Level的复制。从5.0开始，MySQL 的复制已经解决了大量老版本中出现的无法正确复制的问题。但是由于存储过程的出现，给 MySQL 的复制又带来了更大的新挑战。另外，看到官方文档说，从5.1.8版本开始，MySQL 提供了除Statement Level和Row Level之外的第三种复制模式：Mixed Level，实际上就是前两种模式的结合。在Mixed模式下，MySQL会根据执行的每一条具体的 Query 语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种。新版本中的Statment level还是和以前一样，仅仅记录执行的语句。而新版本的Mysql中队Row Level模式也被做了优化，并不是所有的修改都会以Row Level来记录，像遇到表结构变更的时候就会以statement模式来记录，如果 Query 语句确实就是 UPDATE 或者 DELETE 等修改数据的语句，那么还是会记录所有行的变更。

13.3 Replication 常用架构

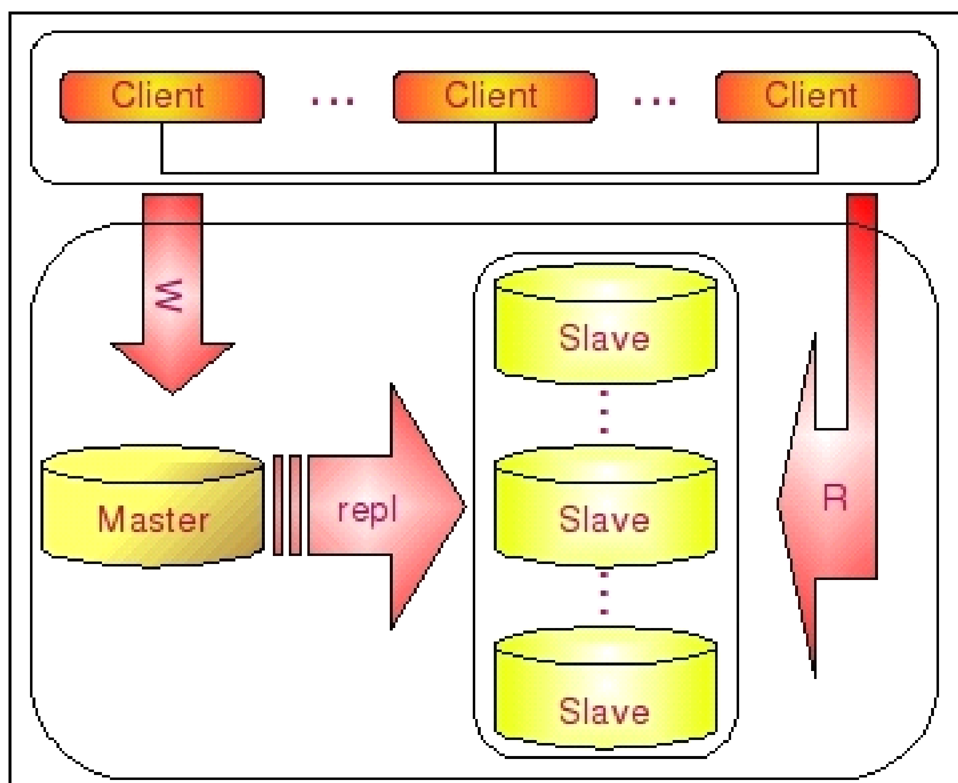
MySQL Replicaion 本身是一个比较简单的架构，就是一台 MySQL 服务器（Slave）从另一台 MySQL 服务器（Master）进行日志的复制然后再解析日志并应用到自身。一个复制环境仅仅只需要两台运行有 MySQL Server 的主机即可，甚至更为简单的时候我们可以在同一台物理服务器主机上面启动两个 mysqld instance，一个作为 Master 而另一个作为Slave 来完成复制环境的搭建。但是在实际应用环境中，我们可以根据实际的业务需求利用MySQL Replication 的功能自己定制搭建出其他多种更利于 Scale Out 的复制架构。如Dual Master 架构，级联复制架构等。下面我们针对比较典型的三种复制架构进行一些相应的分析介绍。

13.3.1 常规复制架构(Master - Slaves)

在实际应用场景中，MySQL 复制90% 以上都是一个Master 复制到一个或者多个Slave的架构模式，主要用于读压力比较大的应用的数据库端廉价扩展解决方案。因为只要

Master 和 Slave 的压力不是太大（尤其是 Slave 端压力）的话，异步复制的延时一般都很少很少。尤其是自从 Slave 端的复制方式改成两个线程处理之后，更是减小了 Slave 端的延时问题。而带来的效益是，对于数据实时性要求不是特别 Critical 的应用，只需要通过廉价的 pc server 来扩展 Slave 的数量，将读压力分散到多台 Slave 的机器上面，即可通过分散单台数据库服务器的读压力来解决数据库端的读性能瓶颈，毕竟在大多数数据库应用系统中的读压力还是要比写压力大很多。这在很大程度上解决了目前很多中小型网站的数据库压力瓶颈问题，甚至有些大型网站也在使用类似方案解决数据库瓶颈。

这个架构可以通过下图比较清晰的展示：



一个 Master 复制多个 Slave 的架构实施非常简单，多个 Slave 和单个 Slave 的实施并没有实质性的区别。在 Master 端并不 Care 有多少个 Slave 连上了自己，只要有 Slave 的 IO 线程通过了连接认证，向他请求指定位置之后的 Binary Log 信息，他就会按照该 IO 线程的要求，读取自己的 Binary Log 信息，返回给 Slave 的 IO 线程。

大家应该都比较清楚，从一个 Master 节点可以复制出多个 Slave 节点，可能有人会想，那一个 Slave 节点是否可以从多个 Master 节点上面进行复制呢？至少在目前来看，MySQL 是做不到的，以后是否会支持就不清楚了。

MySQL 不支持一个 Slave 节点从多个 Master 节点来进行复制的架构，主要是为了避免冲突的问题，防止多个数据源之间的数据出现冲突，而造成最后数据的不一致性。不过听说已经有人开发了相关的 patch，让 MySQL 支持一个 Slave 节点从多个 Master 结点作为数据源来进行复制，这也正是 MySQL 开源的性质所带来的好处。

对于 Replication 的配置细节，在 MySQL 的官方文档上面已经说的非常清楚了，甚至介绍了多种实现 Slave 的配置方式，在下一节中我们也会通过一个具体的示例来演示搭建一个 Replication 环境的详细过程以及注意事项。

13.3.2 Dual Master 复制架构(Master - Master)

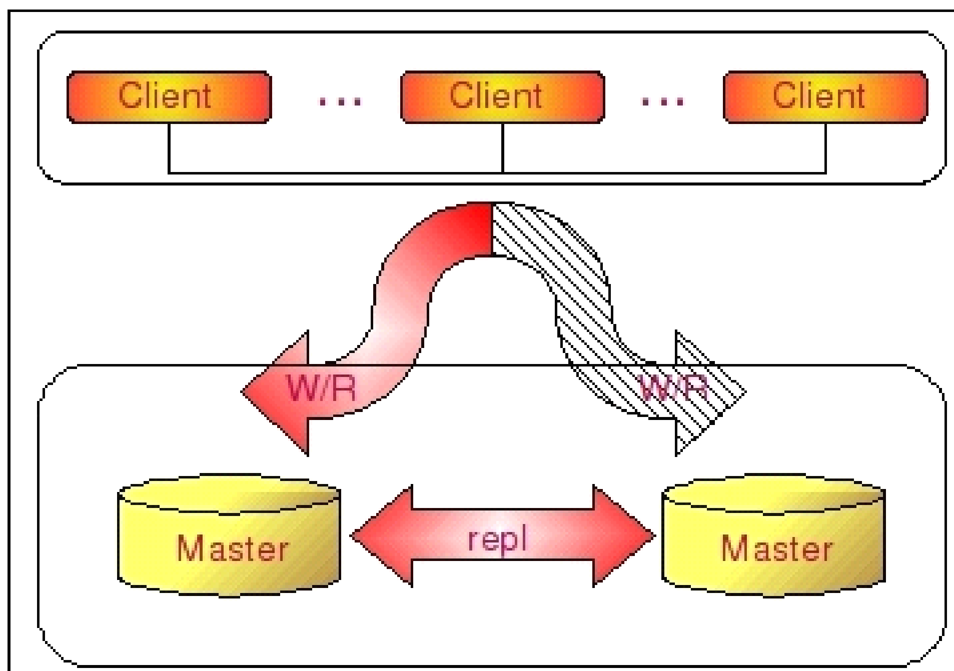
有些时候，简单的从一个 MySQL 复制到另外一个 MySQL 的基本 Replication 架构，可能还会需要在一些特定的场景下进行 Master 的切换。如在 Master 端需要进行一些特别的维护操作的时候，可能需要停 MySQL 的服务。这时候，为了尽可能减少应用系统写服务的停机时间，最佳的做法就是将我们的 Slave 节点切换成 Master 来提供写入的服务。

但是这样一来，我们原来 Master 节点的数据就会和实际的数据不一致了。当原 Master 启动可以正常提供服务的时候，由于数据的不一致，我们就不得不通过反转原 Master - Slave 关系，重新搭建 Replication 环境，并以原 Master 作为 Slave 来对外提供读的服务。重新搭建 Replication 环境会给我们带来很多额外的工作量，如果没有合适的备份，可能还会让 Replication 的搭建过程非常麻烦。

为了解决这个问题，我们可以通过搭建 Dual Master 环境来避免很多的问题。何谓 Dual Master 环境？实际上就是两个 MySQL Server 互相将对方作为自己的 Master，自己作为对方的 Slave 来进行复制。这样，任何一方所做的变更，都会通过复制应用到另外一方的数据库中。

可能有些读者朋友会有一个担心，这样搭建复制环境之后，难道不会造成两台 MySQL 之间的循环复制么？实际上 MySQL 自己早就想到了这一点，所以在 MySQL 的 Binary Log 中记录了当前 MySQL 的 server-id，而且这个参数也是我们搭建 MySQL Replication 的时候必须明确指定，而且 Master 和 Slave 的 server-id 参数值比需要不一致才能使 MySQL Replication 搭建成功。一旦有了 server-id 的值之后，MySQL 就很容易判断某个变更是从哪一个 MySQL Server 最初产生的，所以就很容易避免出现循环复制的情况。而且，如果我们不打开记录 Slave 的 Binary Log 的选项（--log-slave-update）的时候，MySQL 根本就不会记录复制过程中的变更到 Binary Log 中，就更不用担心可能会出现循环复制的情形了。

下如将更清晰的展示 Dual Master 复制架构组成：



通过 Dual Master 复制架构，我们不仅能够避免因为正常的常规维护操作需要的停机所带来的重新搭建 Replication 环境的操作，因为我们任何一端都记录了自己当前复制到对方的什么位置了，当系统起来之后，就会自动开始从之前的位置重新开始复制，而不需要人为去进行任何干预，大大节省了维护成本。

不仅仅如此，Dual Master 复制架构和一些第三方的 HA 管理软件结合，还可以在我们当前正在使用的 Master 出现异常无法提供服务之后，非常迅速的自动切换另外一端来提供相应的服务，减少异常情况下带来的停机时间，并且完全不需要人工干预。

当然，我们搭建成一个 Dual Master 环境，并不是为了让两端都提供写的服务。在正常情况下，我们都只会将其中一端开启写服务，另外一端仅仅只是提供读服务，或者完全不提供任何服务，仅仅只是作为一个备用的机器存在。为什么我们一般都只开启其中的一端来提供写服务呢？主要还是为了避免数据的冲突，防止造成数据的不一致性。因为即使在两边执行的修改有先后顺序，但由于 Replication 是异步的实现机制，同样会导致即使晚做的修改也可能会被早做的修改所覆盖，就像如下情形：

时间点	MySQL A	MySQL B
1	更新 x 表 y 记录为 10	
2		更新 x 表 y 记录为 20
3		获取到 A 日志并应用，更新 x 表的 y 记录为 10（不符合期望）
4	获取 B 日志更新 x 表 y 记录为 20（符合期望）	

这中情形下，不仅在 B 库上面的数据不是用户所期望的结果，A 和 B 两边的数据也出现了不一致。

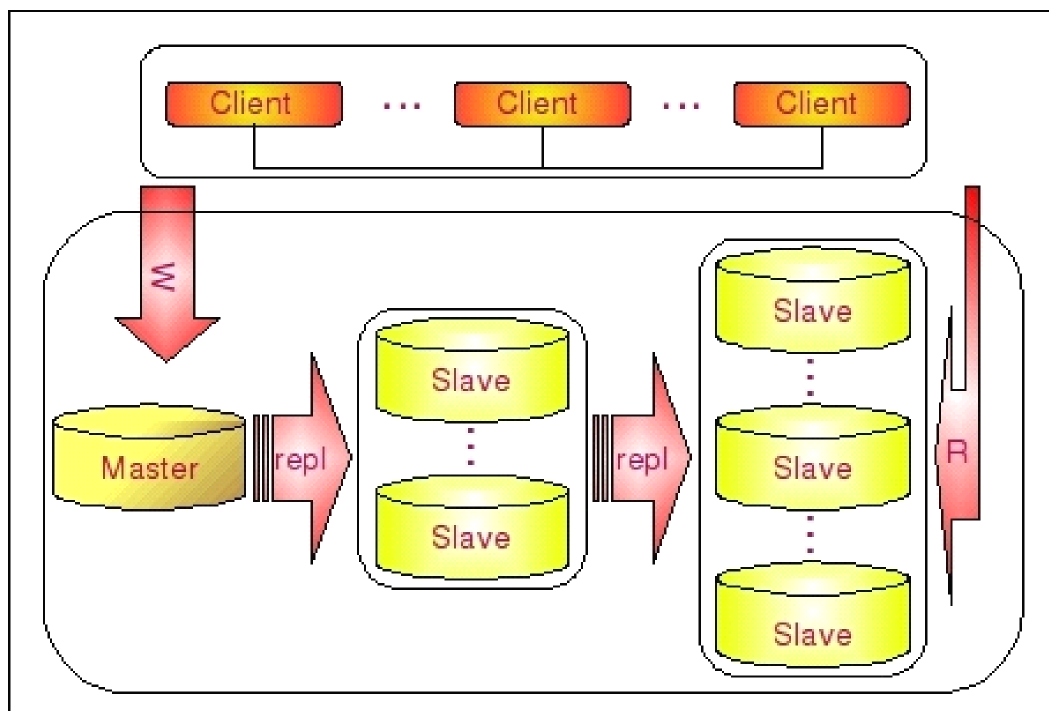
当然，我们也可以通过特殊的约定，让某些表的写操作全部在一端，而另外一些表的写操作全部在另外一端，保证两端不会操作相同的表，这样就能避免上面问题的发生了。

13.3.3 级联复制架构(Master - Slaves - Slaves ...)

在有些应用场景中，可能读写压力差别比较大，读压力特别的大，一个 Master 可能需要上 10 台甚至更多的 Slave 才能够支撑注读的压力。这时候，Master 就会比较吃力了，因为仅仅连上来的 Slave IO 线程就比较多了，这样写的压力稍微大一点的时候，Master 端因为复制就会消耗较多的资源，很容易造成复制的延时。

遇到这种情况如何解决呢？这时候我们就可以利用 MySQL 可以在 Slave 端记录复制所产生变更的 Binary Log 信息的功能，也就是打开 —log-slave-update 选项。然后，通过二级（或者是更多级别）复制来减少 Master 端因为复制所带来的压力。也就是说，我们首先通过少数几台 MySQL 从 Master 来进行复制，这几台机器我们姑且称之为第一级 Slave 集群，然后其他的 Slave 再从第一级 Slave 集群来进行复制。从第一级 Slave 进行复制的 Slave，我称之为第二级 Slave 集群。如果有需要，我们可以继续往下增加更多层次的复制。这样，我们很容易就控制了每一台 MySQL 上面所附属 Slave 的数量。这种架构我称之为 Master - Slaves - Slaves 架构

这种多层级联复制的架构，很容易就解决了 Master 端因为附属 Slave 太多而成为瓶颈的风险。下图展示了多层级联复制的 Replication 架构。



当然，如果条件允许，我更倾向于建议大家通过拆分成多个 Replication 集群来解决

上述瓶颈问题。毕竟 Slave 并没有减少写的量，所有 Slave 实际上仍然还是应用了所有的数据变更操作，没有减少任何写 IO。相反，Slave 越多，整个集群的写 IO 总量也就会越多，我们没有非常明显的感觉，仅仅只是因为分散到了多台机器上面，所以不是很容易表现出来。

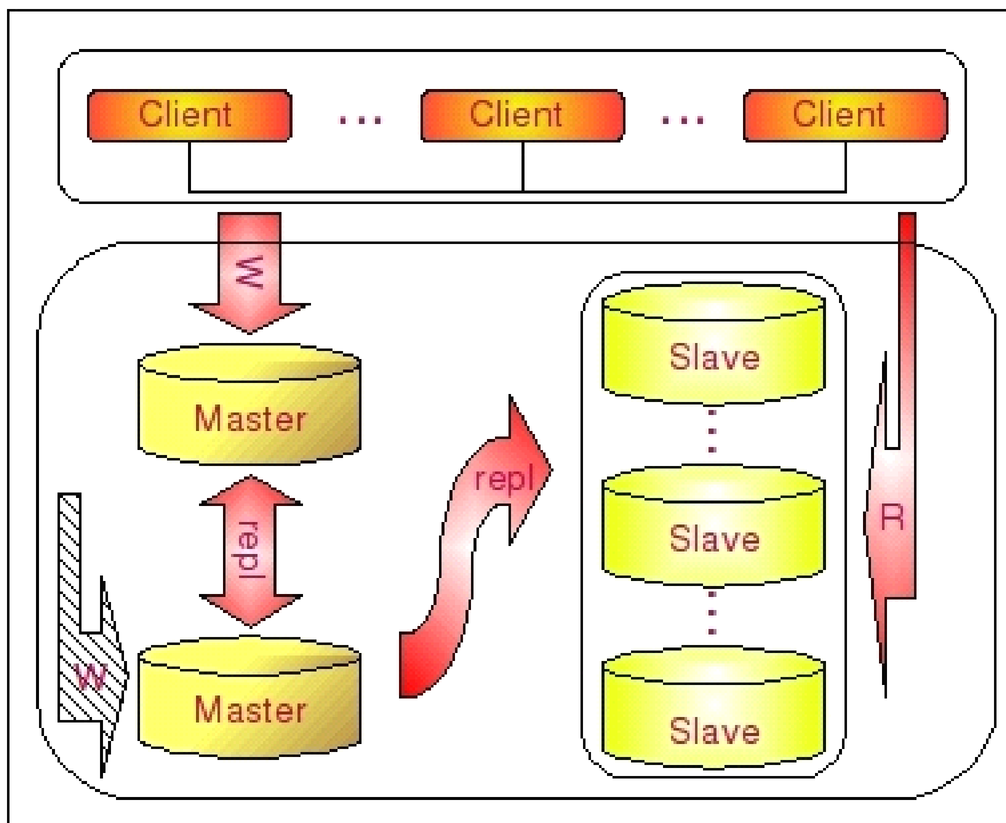
此外，增加复制的级联层次，同一个变更传到最底层的 Slave 所需要经过的 MySQL 也会更多，同样可能造成延时较长的风险。

而如果我们通过分拆集群的方式来解决的话，可能就会要好很多了，当然，分拆集群也需要更复杂的技术和更复杂的应用系统架构。

13.3.4 Dual Master 与级联复制结合架构(Master - Master - Slaves)

级联复制在一定程度上面确实解决了 Master 因为所附属的 Slave 过多而成为瓶颈的问题，但是他并不能解决人工维护和出现异常需要切换后可能存在重新搭建 Replication 的问题。这样就很自然的引申出了 Dual Master 与级联复制结合的 Replication 架构，我称之为 Master - Master - Slaves 架构

和 Master - Slaves - Slaves 架构相比，区别仅仅只是将第一级 Slave 集群换成了一台单独的 Master，作为备用 Master，然后再从这个备用的 Master 进行复制到一个 Slave 集群。下面的图片更清晰的展示了这个架构的组成：



这种 Dual Master 与级联复制结合的架构，最大的好处就是既可以避免主 Master 的写入操作不会受到 Slave 集群的复制所带来的影响，同时主 Master 需要切换的时候也基本上不会出现重搭 Replication 的情况。但是，这个架构也有一个弊端，那就是备用的 Master 有可能成为瓶颈，因为如果后面的 Slave 集群比较大的话，备用 Master 可能会因为过多的 Slave IO 线程请求而成为瓶颈。当然，该备用 Master 不提供任何的读服务的时候，瓶颈出现的可能性并不是特别高，如果出现瓶颈，也可以在备用 Master 后面再次进行级联复制，架设多层 Slave 集群。当然，级联复制的级别越多，Slave 集群可能出现的数据延时也会更为明显，所以考虑使用多层级联复制之前，也需要评估数据延时对应用系统的影响。

13.4 Replication 搭建实现

MySQL Replication 环境的搭建实现比较简单，总的来说其实就是四步，第一步是做好 Master 端的准备工作。第二步是取得 Master 端数据的“快照”备份。第三步则是在 Slave 端恢复 Master 的备份“快照”。第四步就是在 Slave 端设置 Master 相关配置，然后启动复制。在这一节中，并不是列举一个搭建 Replication 环境的详细过程，因为这在 MySQL 官方操作手册中已经有较为详细的描述了，我主要是针对搭建环境中几个主要的操作步骤中可以使用的各种实现方法的介绍，下面我们针对这四步操作及需要注意的地方进行一个简单的分析。

1. Master 端准备工作

在搭建 Replication 环境之前，首先要保证 Master 端 MySQL 记录 Binary Log 的选项打开，因为 MySQL Replication 就是通过 Binary Log 来实现的。让 Master 端 MySQL 记录 Binary Log 可以在启动 MySQL Server 的时候使用 —log-bin 选项或者在 MySQL 的配置文件 my.cnf 中配置 log-bin[=path for binary log]参数选项。

在开启了记录 Binary Log 功能之后，我们还需要准备一个用于复制的 MySQL 用户。可以通过给一个现有帐户授予复制相关的权限，也可以创建一个全新的专用于复制的帐户。当然，我还是建议用一个专用于复制的帐户来进行复制。在之前“MySQL 安全管理”部分也已经介绍过了，通过特定的帐户处理特定一类的工作，不论是在安全策略方面更有利，对于维护来说也有更大的便利性。实现 MySQL Replication 仅仅只需要“REPLICATION SLAVE”权限即可。可以通过如下方式来创建这个用户：

```
root@localhost : mysql 04:16:18> CREATE USER 'repl'@'192.168.0.2'
    -> IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.00 sec)

root@localhost : mysql 04:16:34> GRANT REPLICATION SLAVE ON *.*
    -> TO 'repl'@'192.168.0.2';
Query OK, 0 rows affected (0.00 sec)
```

这里首先通过 CREATE USER 命令创建了一个仅仅具有最基本权限的用户 repl，然后再通过 GRANT 命令授予该用户 REPLICATION SLAVE 的权限。当然，我们也可以仅仅执行上面的第二条命令，即可创建出我们所需的用户，这已经在“MySQL 安全管理”部分介绍过了。

2. 获取 Master 端的备份“快照”

这里所说的 Master 端的备份“快照"，并不是特指通过类似 LVM 之类的软件所做的 snapshot，而是所有数据均是基于某一特定时刻的，数据完整性和一致性都可以得到保证的备份集。同时还需要取得该备份集时刻所对应的 Master 端 Binary Log 的准确 Log Position，因为在后面配置 Slave 的时候会用到。

一般来说，我们可以通过如下集中办法获得一个具有一致性和完整性的备份集以及所对应的 Log Position：

◆ 通过数据库全库冷备份

对于可以停机的数据库，我们可以通过关闭 Master 端 MySQL，然后通过 copy 所有数据文件和日志文件到需要搭建 Slave 的主机中合适的位置，这样所得到的备份集是最完整的。在做完备份之后，然后再启动 Master 端的 MySQL。

当然，这样我们还仅仅只是得到了一个满足要求的备份集，我们还需要这个备份集所对应的日志位置才能可以。对于这样的备份集，我们有多种方法可以获取到对应的日志位置。如在 Master 刚刚启动之后，还没有应用程序连接上 Master 之前，通过执行 SHOW Master STATUS 命令从 Master 端获取到我们可以使用的 Log Position。如果我

们无法在 Master 启动之后控制应用程序的连接，那么可能在我们还没有来得及执行 SHOW Master STATUS 命令之前就已经有数据写进来了，这时候我们可以通过 mysqlbinlog 客户端程序分析 Master 最新的一个 Binary Log 来获取其第一个有效的 Log Position。当然，如果你非常清楚你所使用的 MySQL 版本每一个新的 Binary Log 第一个有效的日志位置，自然就不需要进行任何操作就可以。

◆ 通过 LVM 或者 ZFS 等具有 snapshot 功能的软件进行“热备份”

如果我们的 Master 是一个需要满足 365 * 24 * 7 服务的数据库,那么我们就无法通过进行冷备份来获取所需要的备份集。这时候，如果我们的 MySQL 运行在支持 Snapshot 功能的文件系统上面（如 ZFS），或者我们的文件系统虽然不支持 Snapshot，但是我们的文件系统运行在 LVM 上面,那么我们都可以通过相关的命令对 MySQL 的数据文件和日志文件所在的目录就做一个 Snapshot，这样就可以得到了一个基本和全库冷备差不多的备份集。

当然，为了保证我们的备份集数据能够完整且一致，我们需要在进行 Snapshot 过程中通过相关命令（FLUSH TABLES WITH READ LOCK）来锁住所有表的写操作，也包括支持事务的存储引擎中 commit 动作，这样才能真正保证该 Snapshot 的所有数据都完整一致。在做完 Snapshot 之后，我们就可以 UNLOCK TABLES 了。可能有些人会担心，如果锁住了所有的写操作，那我们的应用不是就无法提供写服务了么？确实，这是无法避免的，不过，一般来说 Snapshot 操作所需要的时间大都比较短，所以不会影响太长时间。

那 Log Position 怎么办呢？是的，通过 Snapshot 所做的备份，同样需要一个该备份所对应的 Log Position 才能满足搭建 Replication 环境的要求。不过，这种方式下，我们可以比进行冷备份更容易获取到对应的 Log Position。因为从我们锁定了所有表的写入操作开始到解锁之前，数据库不能进行任何写入操作，这个时间段之内任何时候通过执行 SHOW MASTER STATUS 明令都可以得到准确的 Log Position。

由于这种方式在实施过程中并不需要完全停掉 Master 来进行,仅仅只需要停止写入才做，所以我们也可以称之为“热备份"。

◆ 通过 mysqldump 客户端程序

如果我们的数据库不能停机进行冷备份，而且 MySQL 也没有运行在可以进行 Snapshot 的文件系统或者管理软件之上，那么我们就需要通过 mysqldump 工具来将 Master 端需要复制的数据库（或者表）的数据 dump 出来。为了让我们的备份集具有一致性和完整性，我们必须让 dump 数据的这个过程处于同一个事务中，或者锁住所有需要复制的表的写操作。要做到这一点，如果我们使用的是支持事务的存储引擎（如 Innodb),我们可以在执行 mysqldump 程序的时候通过添加 —single-transaction 选项来做到，但是如果我们的存储引擎并不支持事务，或者是需要 dump 表仅仅只有部分支持事务的时候，我们就只能先通过 FLUSH TABLES WITH READ LOCK 命令来暂停所有写入服务，然后再 dump 数据。当然，如果我们仅仅只需要 dump 一个表的数据，就不需要这么麻烦了，因为 mysqldump 程序在 dump 数据的时候实际上就是每个表通过一条 SQL 来得到数据的，所以单个表的时候总是可以保证所取数据的一致性的。

上面的操作我们还只是获得了合适的备份集，还没有该备份集所对应的 Log Position，所以还不能完全满足搭建 Slave 的要求。幸好 mysqldump 程序的开发者早就考虑到这个问题了，所以给 mysqldump 程序增加了另外一个参数选项来帮助我们获取到对应的 Log Position，这个参数选项就是 —master-data 。当我们添加这个参数选项之后，mysqldump 会在 dump 文件中产生一条 CHANGE MASTER TO 命令，命令中记录了 dump 时刻所对应的详细的 Log Position 信息。如下：

测试 dump example 数据库下的 group_message 表：

```
sky@sky:~$ mysqldump --master-data -usky -p example group_message >
group_message.sql
Enter password:
```

然后通过 grep 命令来查找一下看看：

```
sky@sky:~$ grep "CHANGE MASTER" group_message.sql
CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin.000035', MASTER_LOG_POS=399;
```

连 CHANGE MASTER TO 的命令都已经给我们准备好了，还真够体贴的，呵呵。

如果我们是要一次性 dump 多个支持事务的表的时候，可能很多人会选择通过添加—single-transaction 选项来保证数据的一致性和完整性。这确实是一个不错的选择。但是，如果我们需要 dump 的数据量比较大的时候，可能会产生一个很大的事务，而且会持续较长的时间。

◆ 通过现有某一个 Slave 端进行“热备份”

如果现在已经有 Slave 从我们需要搭建 Replication 环境的 Master 上进行复制的话，那我们这个备份集就非常容易取得了。我们可以暂时性的停掉现有 Slave（如果有多台则仅仅只需要停止其中的一台），同时执行一次 FLUSH TABLES 命令来刷新所有表和索引的数据。这时候在该 Slave 上面就不会再有任何的写入操作了，我们既可以通过 copy 所有的数据文件和日志文件来做一个全备份，同时也可以通过 Snapshot（如果支持）来进行备份。当然，如果支持 Snapshot 功能，还是建议大家通过 Snapshot 来做，因为这样可以使 Slave 停止复制的时间大大缩短，减少该 Slave 的数据延时。

通过现有 Slave 来获取备份集的方式，不仅仅得到数据库备份的方式很简单，连所需要 Log Position，甚至是新 Slave 后期的配置等相关动作都可以省略掉，只需要新的 Slave 完全基于这个备份集来启动，就可以正常从 Master 进行复制了。

整个过程中我们仅仅只是在短暂时间内停止了某台现有 Slave 的复制线程，对系统的正常服务影响很小，所以这种方式也基本可以称之为“热备份"。

◆ 通过

3. Slave 端恢复备份“快照”

上面第二步我们已经获取到了所需要的备份集了，这一步所需要做的就是将上一步所得到的备份集恢复到我们的 Slave 端的 MySQL 中。

针对上面四种方法所获取的备份集的不同，在 Slave 端的恢复操作也有区别。下面就针对四种备份集的恢复做一个简单的说明：

◆ 恢复全库冷备份集

由于这个备份集是一个完整的数据库物理备份，我们仅仅只需要将这个备份集通过 FTP 或者是 SCP 之类的网络传输软件复制到 Slave 所在的主机，根据 Slave 上 my.cnf 配置文件的设置，将文件存放在相应的目录，覆盖现有所有的数据和日志等相关文件，然后再启动 Slave 端的 MySQL，就完成了整个恢复过程。

◆ 恢复对 Master 进行 Snapshot 得到的备份集

对于通过对 Master 进行 Snapshot 所得到的备份集，实际上和全库冷备的恢复方法基本一样，唯一的差别只是首先需要将该 Snapshot 通过相应的文件系统 mount 到某个目录下，然后才能进行后续的文件拷贝操作。之后的相关操作和恢复全库冷备份集基本一致，就不再累述。

◆ 恢复 mysqldump 得到的备份集

通过 mysqldump 客户端程序所得到的备份集，和前面两种备份集的恢复方式有较大的差别。因为前面两种备份集的都属于物理备份，而通过 mysqldump 客户端程序所做的备份属于逻辑备份。恢复 mysqldump 备份集的方式是通过 mysql 客户端程序来执行备份文件中的所有 SQL 语句。

使用 mysql 客户端程序在 Slave 端恢复之前，建议复制出通过 —master-data 所得到的 CHANGE MASTER TO 命令部分，然后在备份文件中注销掉该部分，再进行恢复。因为该命令并不是一个完整的 CHANGE MASTER TO 命令，如果在配置文件（my.cnf）中没有配置 MASTER_HOST,MASTER_USER,MASTER_PASSWORD 这三个参数的时候，该语句是无法有效完成的。

通过 mysql 客户端程序来恢复备份的方式如下：

```
sky@sky:~$ mysql -u sky -p -Dexample < group_message.sql
```

这样即可将之前通过 mysqldump 客户端程序所做的逻辑备份集恢复到数据库中了。

◆ 恢复通过现有 Slave 所得到的热备份

通过现有 Slave 所得到的备份集和上面第一种或者第二种备份集也差不多。如果是通过直接拷贝数据和日志文件所得到的备份集，那么就和全库冷备一样的备份方式，如果是通过 Snapshot 得到的备份集，就和第二种备份恢复方式完全一致。

4. 配置并启动 Slave

在完成了前面三个步骤之后， Replication 环境的搭建就只需要最后的一个步骤

了，那就是通过 CHANGE MASTER TO 命令来配置 然后再启动 Slave 了。

CHANGE MASTER TO 命令总共需要设置 5 项内容，分别为：
MASTER_HOST：Master 的主机名（或者 IP 地址）；
MASTER_USER：Slave 连接 Master 的用户名，实际上就是之前所创建的 repl 用户；
MASTER_PASSWORD：Slave 连接 Master 的用户的密码；
MASTER_LOG_FILE：开始复制的日志文件名称；
MASTER_LOG_POS：开始复制的日志文件的位置，也就是在之前介绍备份集过程中一致提到的 Log Position。

下面是一个完整的 CHANGE MASTER TO 命令示例：

```
CHANGE MASTER TO
root@localhost : mysql 08:32:38> CHANGE MASTER TO
    -> MASTER_HOST='192.168.0.1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='password',
    -> MASTER_LOG_FILE='mysql-bin.000035',
    -> MASTER_LOG_POS=399;
```

执行完 CHANGE MASTER TO 命令之后，就可以通过如下命令启动 SLAVE 了：

```
root@localhost : mysql 08:33:49> START SLAVE;
```

至此，我们的 Replication 环境就搭建完成了。读者朋友可以自己进行相应的测试来尝试搭建，如果需要了解 MySQL Replication 搭建过程中更为详细的步骤，可以通过查阅 MySQL 官方手册。

13.5 小结

在实际应用场景中，MySQL Replication 是使用最为广泛的一种提高系统扩展性的设计手段。众多的 MySQL 使用者通过 Replication 功能提升系统的扩展性之后，通过简单的增加价格低廉的硬件设备成倍甚至成数量级的提高了原有系统的性能，是广大 MySQL 中低端使用者最为喜爱的功能之一，也是大量 MySQL 使用者选择 MySQL 最为重要的理由之一。

第 14 章 可扩展性设计之数据切分

前言

通过 MySQL Replication 功能所实现的扩展总是会受到数据库大小的限制，一旦数据库过于庞大，尤其是当写入过于频繁，很难由一台主机支撑的时候，我们还是会面临到扩展瓶颈。这时候，我们就必须许找其他技术手段来解决这个瓶颈，那就是我们这一章所要介绍恶的数据切分技术。

14.1 何谓数据切分

可能很多读者朋友在网上或者杂志上面都已经多次见到关于数据切分的相关文章了，只

不过在有些文章中称之为数据的 Sharding。其实不管是称之为数据的 Sharding 还是数据的切分，其概念都是一样的。简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。数据的切分同时还可以提高系统的总体可用性，因为单台设备 Crash 之后，只有总体数据的某部分不可用，而不是所有的数据。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面，这种切分称之为数据的水平（横向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分于垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较根据表名来拆分更为复杂，后期的数据维护也会更为复杂一些。

当我们某个（或者某些）表的数据量和访问量特别的大，通过垂直切分将其放在独立的设备上后仍然无法满足性能要求，这时候我们就必须将垂直切分和水平切分相结合，先垂直切分，然后再水平切分，才能解决这种超大型表的性能问题。

下面我们就针对垂直、水平以及组合切分这三种数据切分方式的架构实现及切分后数据的整合进行相应的分析。

14.2 数据的垂直切分

我们先来看一下，数据的垂直切分到底是如何一个切分法的。数据的垂直切分，也可以称之为纵向切分。将数据库想象成为由很多个一大块一大块的“数据块"（表）组成，我们垂直的将这些“数据块”切开，然后将他们分散到多台数据库主机上面。这样的切分方法就是一个垂直（纵向）的数据切分。

一个架构设计较好的应用系统，其总体功能肯定是由很多个功能模块所组成的，而每一个功能模块所需要的数据对应到数据库中就是一个或者多个表。而在架构设计中，各个功能模块相互之间的交互点越统一越少，系统的耦合度就越低，系统各个模块的维护性以及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。

当我们的功能模块越清晰，耦合度越低，数据垂直切分的规则定义也就越容易。完全可以根据功能模块来进行数据的切分，不同功能模块的数据存放于不同的数据库主机中，可以很容易就避免掉跨数据库的 Join 存在，同时系统架构也非常的清晰。

当然，很难有系统能够做到所有功能模块所使用的表完全独立，完全不需要访问对方的表或者需要两个模块的表进行 Join 操作。这种情况下，我们就必须根据实际的应用场景进行评估权衡。决定是迁就应用程序将需要 Join 的表的相关某快都存放在同一个数据库中，还是让应用程序做更多的事情，也就是程序完全通过模块接口取得不同数据库中的数据，然后在程序中完成 Join 操作。

一般来说，如果是一个负载相对不是很大的系统，而且表关联又非常的频繁，那可能数据库让步，将几个相关模块合并在一起减少应用程序的工作的方案可以减少较多的工作量，是一个可行的方案。

当然，通过数据库的让步，让多个模块集中共用数据源，实际上也是简介的默许了各模块架构耦合度增大的发展，可能会让以后的架构越来越恶化。尤其是当发展到一定阶段之后，发现数据库实在无法承担这些表所带来的压力，不得不面临再次切分的时候，所带来的架构改造成本可能会远远大于最初的时候。

所以，在数据库进行垂直切分的时候，如何切分，切分到什么样的程度，是一个比较考验人的难题。只能在实际的应用场景中通过平衡各方面的成本和收益，才能分析出一个真正适合自己的拆分方案。

比如在本书所使用示例系统的 example 数据库，我们简单的分析一下，然后再设计一个简单的切分规则，进行一次垂直垂直拆分。

系统功能可以基本分为四个功能模块：用户，群组消息，相册以及事件，分别对应为如下这些表：

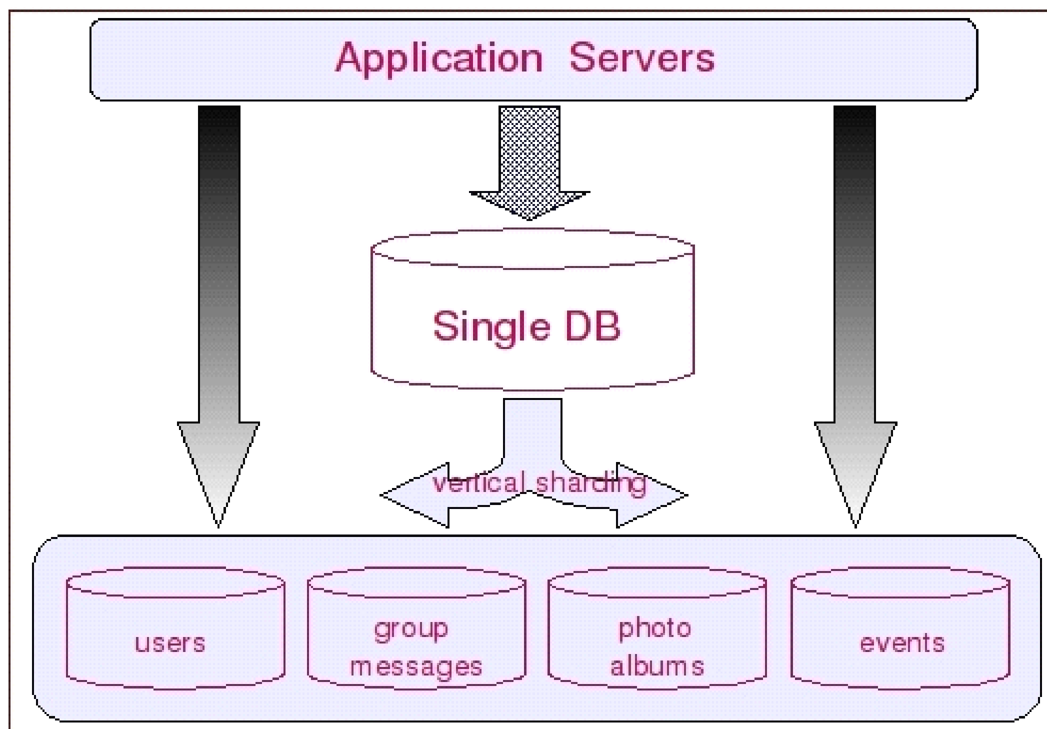
1. 用户模块表：user,user_profile,user_group,user_photo_album
2. 群组讨论表：groups,group_message,group_message_content,top_message
3. 相册相关表：photo,photo_album,photo_album_relation,photo_comment
4. 事件信息表：event

初略一看，没有哪一个模块可以脱离其他模块独立存在，模块与模块之间都存在着关系，莫非无法切分？

当然不是，我们再稍微深入分析一下，可以发现，虽然各个模块所使用的表之间都有关联，但是关联关系还算比较清晰，也比较简单。

◆ 群组讨论模块和用户模块之间主要存在通过用户或者是群组关系来进行关联。一般关联的时候都会是通过用户的 id 或者 nick_name 以及 group 的 id 来进行关联，通过模块之间的接口实现不会带来太多麻烦；
◆ 相册模块仅仅与用户模块存在通过用户的关联。这两个模块之间的关联基本就有通过用户 id 关联的内容，简单清晰，接口明确；
◆ 事件模块与各个模块可能都有关联，但是都只关注其各个模块中对象的 ID 信息，同样可以做到很容易分拆。

所以，我们第一步可以将数据库按照功能模块相关的表进行一次垂直拆分，每个模块所涉及的表单独到一个数据库中，模块与模块之间的表关联都在应用系统端通过藉口来处理。如下图所示：



通过这样的垂直切分之后，之前只能通过一个数据库来提供的服务，就被分拆成四个数据库来提供服务，服务能力自然是增加几倍了。

垂直切分的优点

- 数据库的拆分简单明了，拆分规则明确；
- 应用程序模块清晰明确，整合容易；
- 数据维护方便易行，容易定位；

垂直切分的缺点

- 部分表关联无法在数据库级别完成，需要在程序中完成；
- 对于访问极其频繁且数据量超大的表仍然存在性能平静，不一定能满足要求；
- 事务处理相对更为复杂；
- 切分达到一定程度之后，扩展性会遇到限制；
- 过读切分可能会带来系统过渡复杂而难以维护。

针对于垂直切分可能遇到数据切分及事务问题，在数据库层面实在是很难找到一个较好的处理方案。实际应用案例中，数据库的垂直切分大多是与应用系统的模块相对应，同一个模块的数据源存放于同一个数据库中，可以解决模块内部的数据关联问题。而模块与模块之间，则通过应用程序以服务接口方式来相互提供所需要的数据。虽然这样做在数据库的总体操作次数方面确实会有所增加，但是在系统整体扩展性以及架构模块化方面，都是有益的。

可能在某些操作的单次响应时间会稍有增加，但是系统的整体性能很可能反而会有一定的提升。而扩展瓶颈问题，就只能依靠下一节将要介绍的数据水平切分架构来解决了。

14.3 数据的水平切分

上面一节分析介绍了数据的垂直切分，这一节再分析一下数据的水平切分。数据的垂直切分基本上可以简单的理解为按照表按照模块来切分数据，而水平切分就不再是按照表或者是功能模块来切分了。一般来说，简单的水平切分主要是将某个访问极其平凡的表再按照某个字段的某种规则来分散到多个表之中，每个表中包含一部分数据。

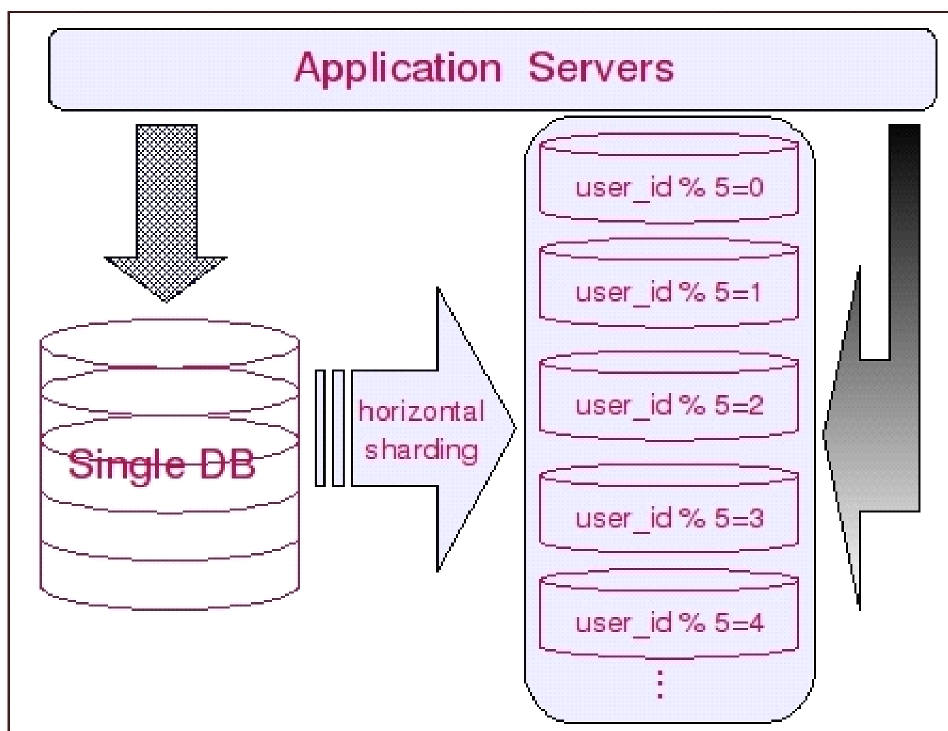
简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中。当然，为了能够比较容易的判定各行数据被切分到哪个数据库中了，切分总是都需要按照某种特定的规则来进行的。如根据某个数字类型字段基于特定数目取模，某个时间类型字段的范围，或者是某个字符类型字段的 hash 值。如果整个系统中大部分核心表都可以通过某个字段来进行关联，那这个字段自然是一个进行水平分区的上上之选了，当然，非常特殊无法使用就只能另选其他了。

一般来说，像现在互联网非常火爆的 Web2.0 类型的网站，基本上大部分数据都能够通过会员用户信息关联上，可能很多核心表都非常适合通过会员 ID 来进行数据的水平切分。而像论坛社区讨论系统，就更容易切分了，非常容易按照论坛编号来进行数据的水平切分。切分之后基本上不会出现各个库之间的交互。

如我们的示例系统，所有数据都是和用户关联的，那么我们就可以根据用户来进行水平拆分，将不同用户的数据切分到不同的数据库中。当然，唯一有点区别的是用户模块中的 groups 表和用户没有直接关系，所以 groups 不能根据用户来进行水平拆分。对于这种特殊情况下的表，我们完全可以独立出来，单独放在一个独立的数据库中。其实这个做法可以说是利用了前面一节所介绍的“数据的垂直切分”方法，我将在下一节中更为详细的介绍这种垂直切分与水平切分同时使用的联合切分方法。

所以，对于我们的示例数据库来说，大部分的表都可以根据用户 ID 来进行水平的切分。不同用户相关的数据进行切分之后存放在不同的数据库中。如将所有用户 ID 通过 2 取模然后分别存放于两个不同的数据库中。每个和用户 ID 关联上的表都可以这样切分。这样，基本上每个用户相关的数据，都在同一个数据库中，即使是需要关联，也可以非常简单的关联上。

我们可以通过下图来更为直观的展示水平切分相关信息：



水平切分的优点

- 表关联基本能够在数据库端全部完成；
- 不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；
- 应用程序端整体架构改动相对较少；
- 事务处理相对简单；
- 只要切分规则能够定义好，基本上较难遇到扩展性限制；

水平切分的缺点

- 切分规则相对更为复杂，很难抽象出一个能够满足整个数据库的切分规则；
- 后期数据的维护难度有所增加，人为手工定位数据更困难；
- 应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

14.4 垂直与水平联合切分的使用

上面两节内容中，我们分别，了解了“垂直”和“水平”这两种切分方式的实现以及切分之后的架构信息，同时也分析了两种架构各自的优缺点。但是在实际的应用场景中，除了那些负载并不是太大，业务逻辑也相对较简单的系统可以通过上面两种切分方法之一来解决扩展性问题之外，恐怕其他大部分业务逻辑稍微复杂一点，系统负载大一些的系统，都无法通过上面任何一种数据的切分方法来实现较好的扩展性，而需要将上述两种切分方法结合使用，不同的场景使用不同的切分方法。

在这一节中，我将结合垂直切分和水平切分各自的优缺点，进一步完善我们的整体架构，让系统的扩展性进一步提高。

一般来说，我们数据库中的所有表很难通过某一个（或少数几个）字段全部关联起来，所以很难简单的仅仅通过数据的水平切分来解决所有问题。而垂直切分也只能解决部分问题，对于那些负载非常高的系统，即使仅仅只是单个表都无法通过单台数据库主机来承担其负载。我们必须结合“垂直”和“水平”两种切分方式同时使用，充分利用两者的优点，避开其缺点。

每一个应用系统的负载都是一步一步增长上来的，在开始遇到性能瓶颈的时候，大多数架构师和 DBA 都会选择先进行数据的垂直拆分，因为这样的成本最先，最符合这个时期所追求的最大投入产出比。然而，随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时期之后，经过了垂直拆分之后的数据库集群可能又再一次不堪重负，遇到了性能瓶颈。

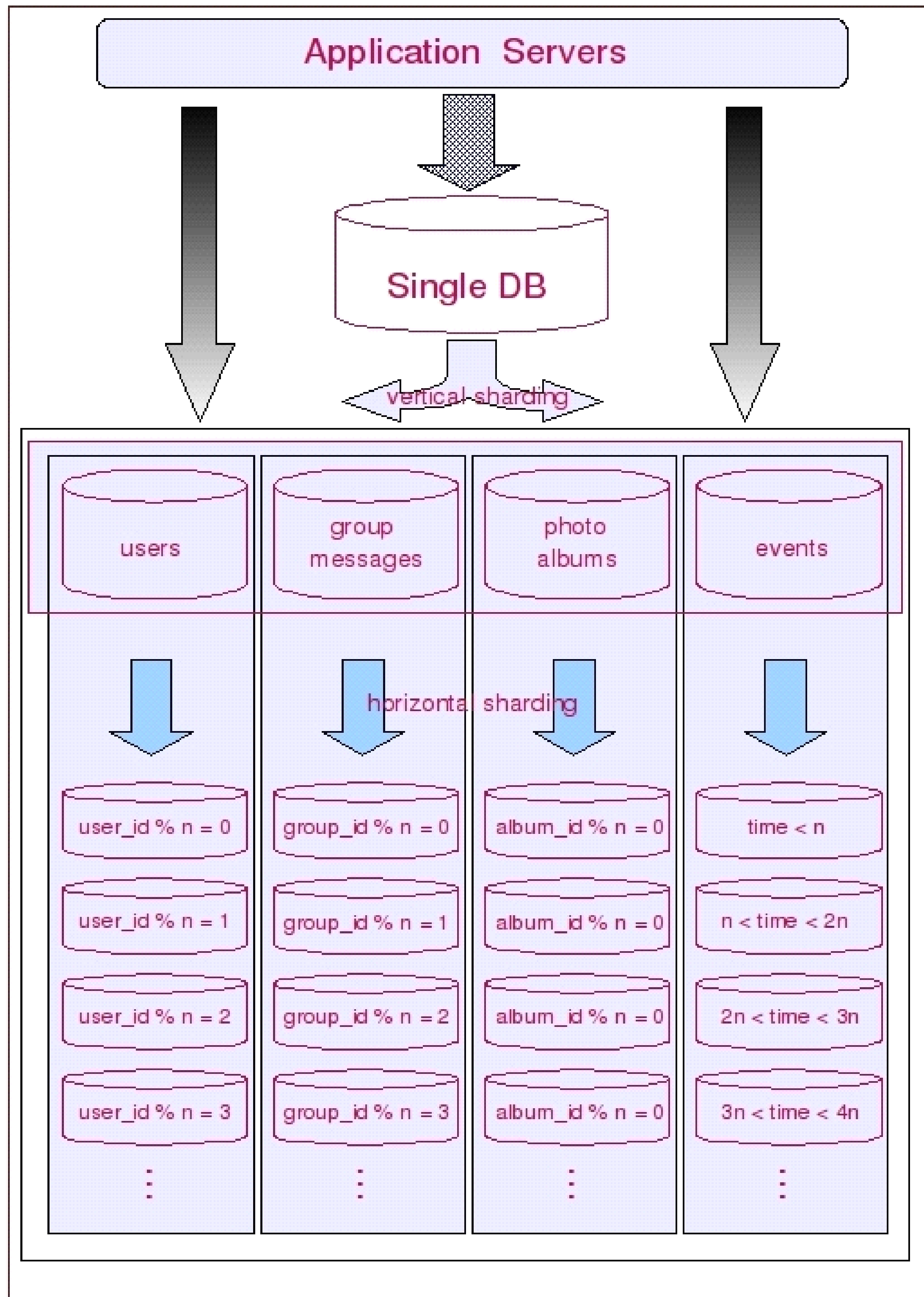
这时候我们该如何抉择？是再次进一步细分模块呢，还是寻求其他的办法来解决？如果我们再一次像最开始那样继续细分模块，进行数据的垂直切分，那我们可能在不久的将来，又会遇到现在所面对的同样的问题。而且随着模块的不断的细化，应用系统的架构也会越来越复杂，整个系统很可能会出现失控的局面。

这时候我们就必须要通过数据的水平切分的优势，来解决这里所遇到的问题。而且，我们完全不必要在使用数据水平切分的时候，推倒之前进行数据垂直切分的成果，而是在其基础上利用水平切分的优势来避开垂直切分的弊端，解决系统复杂性不断扩大的问题。而水平拆分的弊端（规则难以统一）也已经被之前的垂直切分解决掉了，让水平拆分可以进行的得心应手。

对于我们的示例数据库，假设在最开始，我们进行了数据的垂直切分，然而随着业务的不断增长，数据库系统遇到了瓶颈，我们选择重构数据库集群的架构。如何重构？考虑到之前已经做好了数据的垂直切分，而且模块结构清晰明确。而业务增长的势头越来越猛，即使现在进一步再次拆分模块，也坚持不了太久。我们选择了在垂直切分的基础上再进行水平拆分。

在经历过垂直拆分后的各个数据库集群中的每一个都只有一个功能模块，而每个功能模块中的所有表基本上都会与某个字段进行关联。如用户模块全部都可以通过用户 ID 进行切分，群组讨论模块则都通过群组 ID 来切分，相册模块则根据相册 ID 来进切分，最后的事件通知信息表考虑到数据的时限性（仅仅只会访问最近某个事件段的信息），则考虑按时间来切分。

下图展示了切分后的整个架构：



实际上，在很多大型的应用系统中，垂直切分和水平切这两种数据的切分方法基本上都是并存的，而且经常在不断的交替进行，以不断的增加系统的扩展能力。我们在应对不同的应用场景的时候，也需要充分考虑到这两种切分方法各自的局限，以及各自的优势，在不同的时期（负载压力）使用不同的结合方式。

联合切分的优点

- ◆ 可以充分利用垂直切分和水平切分各自的优势而避免各自的缺陷；
- ◆ 让系统扩展性得到最大化提升；

联合切分的缺点

- ◆ 数据库系统架构比较复杂，维护难度更大；
- ◆ 应用程序架构也相对更复杂；

14.5 数据切分及整合方案

通过前面的章节，我们已经很清楚了通过数据库的数据切分可以极大的提高系统的扩展性。但是，数据库中的数据在经过垂直和（或）水平切分被存放在不同的数据库主机之后，应用系统面临的最大问题就是如何来让这些数据源得到较好的整合，可能这也是很多读者朋友非常关心的一个问题。这一节我们主要针对的内容就是分析可以使用的各种可以帮助我们实现数据切分以及数据整合的整体解决方案。

数据的整合很难依靠数据库本身来达到这个效果，虽然 MySQL 存在 Federated 存储引擎，可以解决部分类似的问题，但是在实际应用场景中却很难较好的运用。那我们该如何来整合这些分散在各个 MySQL 主机上面的数据源呢？

总的来说，存在两种解决思路：

1. 在每个应用程序模块中配置管理自己需要的一个（或者多个）数据源，直接访问各个数据库，在模块内完成数据的整合；
2. 通过中间代理层来统一管理所有的数据源，后端数据库集群对前端应用程序透明；

可能 90%以上的人在面对上面这两种解决思路的时候都会倾向于选择第二种，尤其是系统不断变得庞大复杂的时候。确实，这是一个非常正确的选择，虽然短期内需要付出的成本可能会相对更大一些，但是对整个系统的扩展性来说，是非常有帮助的。

所以，对于第一种解决思路我这里就不准备过多的分析，下面我重点分析一下在第二种解决思路中的一些解决方案。

★ 自行开发中间代理层

在决定选择通过数据库的中间代理层来解决数据源整合的架构方向之后，有不少公司（或者企业）选择了通过自行开发符合自身应用特定场景的代理层应用程序。

通过自行开发中间代理层可以最大程度的应对自身应用的特定，最大化的定制很多个性化需求，在面对变化的时候也可以灵活的应对。这应该说是自行开发代理层最大的优势了。

当然，选择自行开发，享受让个性化定制最大化的乐趣的同时，自然也需要投入更多的成本来进行前期研发以及后期的持续升级改进工作，而且本身的技术门槛可能也比简单的 Web 应用要更高一些。所以，在决定选择自行开发之前，还是需要进行比较全面的评估为好。

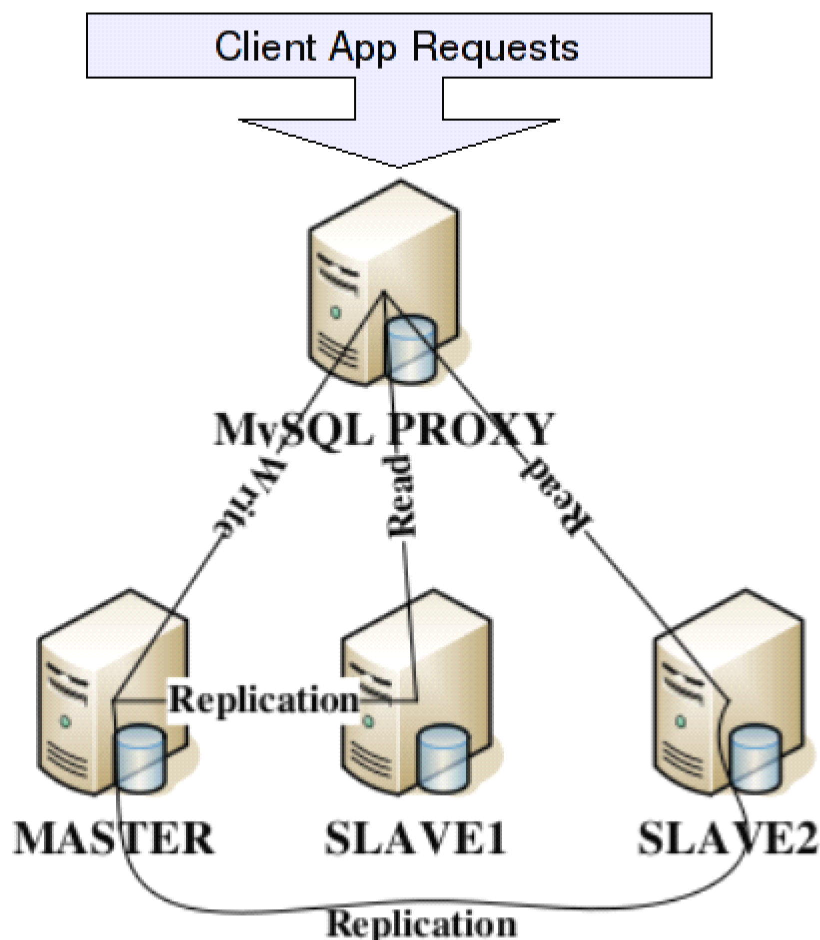
由于自行开发更多时候考虑的是如何更好的适应自身应用系统，应对自身的业务场景，所以这里也不好分析太多。后面我们主要分析一下当前比较流行的几种数据源整合解决方案。

★ 利用 MySQL Proxy 实现数据切分及整合

MySQL Proxy 是 MySQL 官方提供的一个数据库代理层产品，和 MySQL Server 一样，同样是一个基于 GPL 开源协议的开源产品。可用来监视、分析或者传输他们之间的通讯信息。他的灵活性允许你最大限度的使用它，目前具备的功能主要有连接路由，Query 分析，Query 过滤和修改，负载均衡，以及基本的 HA 机制等。

实际上，MySQL Proxy 本身并不具有上述所有的这些功能，而是提供了实现上述功能的基础。要实现这些功能，还需要通过我们自行编写 LUA 脚本来实现。

MySQL Proxy 实际上是在客户端请求与 MySQL Server 之间建立了一个连接池。所有客户端请求都是发向 MySQL Proxy，然后经由 MySQL Proxy 进行相应的分析，判断出是读操作还是写操作，分发至对应的 MySQL Server 上。对于多节点 Slave 集群，也可以起做到负载均衡的效果。以下是 MySQL Proxy 的基本架构图：



通过上面的架构简图，我们可以很清晰的看出 MySQL Proxy 在实际应用中所处的位置，以及能做的基本事情。关于 MySQL Proxy 更为详细的实施细则在 MySQL 官方文档中有非常详细的介绍和示例，感兴趣的读者朋友可以直接从 MySQL 官方网站免费下载或者在线阅读，我这里就不累述浪费纸张了。

★ 利用 Amoeba 实现数据切分及整合

Amoeba 是一个基于 Java 开发的，专注于解决分布式数据库数据源整合 Proxy 程序的开源框架，基于 GPL3 开源协议。目前，Amoeba 已经具有 Query 路由，Query 过滤，读写分离，负载均衡以及 HA 机制等相关内容。

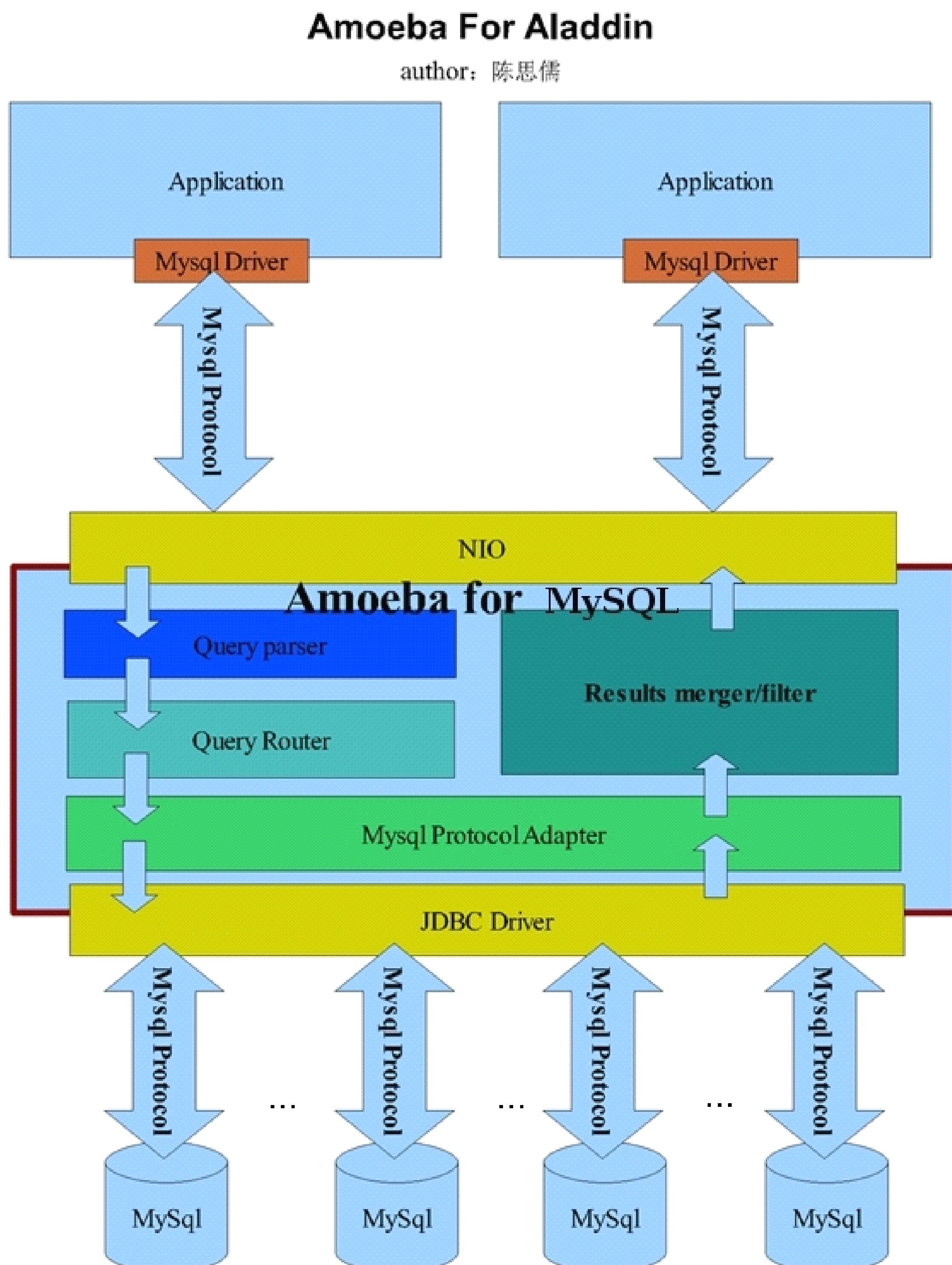
Amoeba 主要解决的以下几个问题：

1. 数据切分后复杂数据源整合；
2. 提供数据切分规则并降低数据切分规则给数据库带来的影响；
3. 降低数据库与客户端的连接数；
4. 读写分离路由；

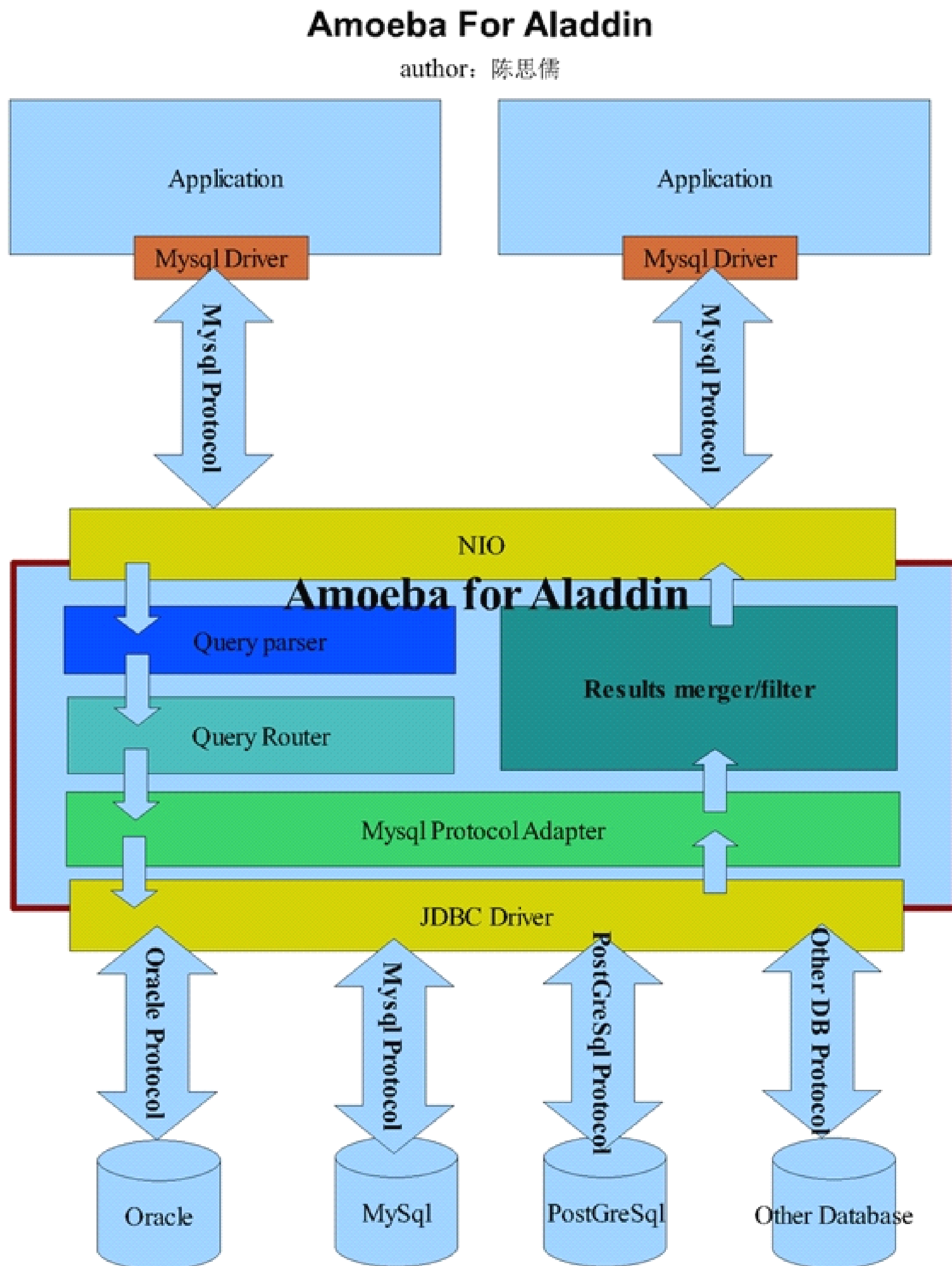
我们可以看出，Amoeba 所做的事情，正好就是我们通过数据切分来提升数据库的扩展性所需要的。

Amoeba 并不是一个代理层的 Proxy 程序，而是一个开发数据库代理层 Proxy 程序的开发框架，目前基于 Amoeba 所开发的 Proxy 程序有 Amoeba For MySQL 和 Amoeba For Aladin 两个。

Amoeba For MySQL 主要是专门针对 MySQL 数据库的解决方案，前端应用程序请求的协议以及后端连接的数据源数据库都必须是 MySQL。对于客户端的任何应用程序来说，Amoeba For MySQL 和一个 MySQL 数据库没有什么区别，任何使用 MySQL 协议的客户端请求，都可以被 Amoeba For MySQL 解析并进行相应的处理。下如可以告诉我们 Amoeba For MySQL 的架构信息（出自 Amoeba 开发者博客）：



Amoeba For Aladin 则是一个适用更为广泛，功能更为强大的 Proxy 程序。他可以同时连接不同数据库的数据源为前端应用程序提供服务，但是仅仅接受符合 MySQL 协议的客户端应用程序请求。也就是说，只要前端应用程序通过 MySQL 协议连接上来之后，Amoeba For Aladin 会自动分析 Query 语句，根据 Query 语句中所请求的数据来自动识别出该所 Query 的数据源是在什么类型数据库的哪一个物理主机上面。下图展示了 Amoeba For Aladin 的架构细节（出自 Amoeba 开发者博客）：



咋一看，两者好像完全一样嘛。细看之后，才会发现两者主要的区别仅在于通过 MySQL Protocal Adapter 处理之后，根据分析结果判断出数据源数据库，然后选择特定的 JDBC

驱动和相应协议连接后端数据库。

其实通过上面两个架构图大家可能也已经发现了 Amoeba 的特点了，他仅仅只是一个开发框架，我们除了选择他已经提供的 For MySQL 和 For Aladin 这两款产品之外，还可以基于自身的需求进行相应的二次开发，得到更适应我们自己应用特点的 Proxy 程序。

当对于使用 MySQL 数据库来说，不论是 Amoeba For MySQL 还是 Amoeba For Aladin 都可以很好的使用。当然，考虑到任何一个系统越是复杂，其性能肯定就会有一定的损失，维护成本自然也会相对更高一些。所以，对于仅仅需要使用 MySQL 数据库的时候，我还是建议使用 Amoeba For MySQL。

Amoeba For MySQL 的使用非常简单，所有的配置文件都是标准的 XML 文件，总共有四个配置文件。分别为：

- amoeba.xml：主配置文件，配置所有数据源以及 Amoeba 自身的参数设置；
- rule.xml：配置所有 Query 路由规则的信息；
- functionMap.xml：配置用于解析 Query 中的函数所对应的 Java 实现类；
- rullFunctionMap.xml：配置路由规则中需要使用到的特定函数的实现类；

如果您的规则不是太复杂，基本上仅需要使用到上面四个配置文件中的前面两个就可完成所有工作。Proxy 程序常用的功能如读写分离，负载均衡等配置都在 amoeba.xml 中进行。此外，Amoeba 已经支持了实现数据的垂直切分和水平切分的自动路由，路由规则可以在 rule.xml 进行设置。

目前 Amoeba 少有欠缺的主要就是其在线管理功能以及对事务的支持了，曾经在与相关开发者的沟通过程中提出过相关的建议，希望能够提供一个可以进行在线维护管理的命令行管理工具，方便在线维护使用，得到的反馈是管理专门的管理模块已经纳入开发日程了。另外在事务支持方面暂时还是 Amoeba 无法做到的，即使客户端应用在提交给 Amoeba 的请求是包含事务信息的，Amoeba 也会忽略事务相关信息。当然，在经过不断完善之后，我相信事务支持肯定是 Amoeba 重点考虑增加的 feature。

关于 Amoeba 更为详细的使用方法读者朋友可以通过 Amoeba 开发者博客（http://amoeba.sf.net）上面提供的使用手册获取，这里就不再细述了。

★ 利用 HiveDB 实现数据切分及整合

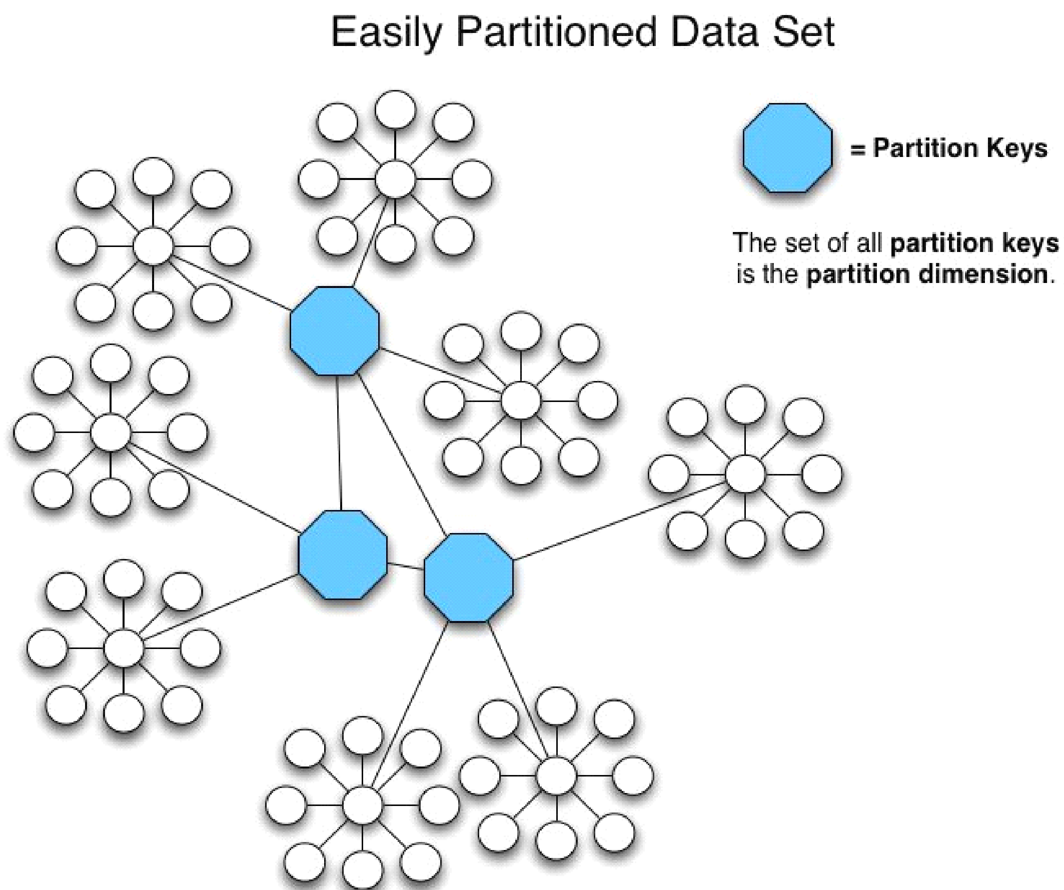
和前面的 MySQL Proxy 以及 Amoeba 一样，HiveDB 同样是一个基于 Java 针对 MySQL 数据库的提供数据切分及整合的开源框架，只是目前的 HiveDB 仅仅支持数据的水平切分。主要解决大数据量下数据库的扩展性及数据的高性能访问问题，同时支持数据的冗余及基本的 HA 机制。

HiveDB 的实现机制与 MySQL Proxy 和 Amoeba 有一定的差异，他并不是借助 MySQL 的 Replication 功能来实现数据的冗余，而是自行实现了数据冗余机制，而其底层主要是基于 Hibernate Shards 来实现的数据切分工作。

在 HiveDB 中，通过用户自定义的各种 Partition keys（其实就是制定数据切分规则），将数据分散到多个 MySQL Server 中。在访问的时候，在运行 Query 请求的时候，会自动分析过滤条件，并行从多个 MySQL Server 中读取数据，并合并结果集返回给客户端应用程序。

单纯从功能方面来讲，HiveDB 可能并不如 MySQL Proxy 和 Amoeba 那样强大，但是其数据切分的思路与前面二者并无本质差异。此外，HiveDB 并不仅仅只是一个开源爱好者所共享的内容，而是存在商业公司支持的开源项目。

下面是 HiveDB 官方网站上面一章图片，描述了 HiveDB 如何来组织数据的基本信息，虽然不能详细的表现出太多架构方面的信息，但是也基本可以展示出其在数据切分方面独特的一面了。



★ 其他实现数据切分及整合的解决方案

除了上面介绍的几个数据切分及整合的整体解决方案之外，还存在很多其他同样提供了数据切分与整合的解决方案。如基于 MySQL Proxy 的基础上做了进一步扩展的 HSCALE，通过 Rails 构建的 Spock Proxy，以及基于 Pathon 的 Pyshards 等等。

不管大家选择使用哪一种解决方案，总体设计思路基本上都不应该会有任何变化，那就是通过数据的垂直和水平切分，增强数据库的整体服务能力，让应用系统的整体扩展能力尽可能的提升，扩展方式尽可能的便捷。

只要我们通过中间层 Proxy 应用程序较好的解决了数据切分和数据源整合问题，那么数据库的线性扩展能力将很容易做到像我们的应用程序一样方便，只需要通过添加廉价的 PC Server 服务器，即可线性增加数据库集群的整体服务能力，让数据库不再轻易成为应用系统的性能瓶颈。

14.6 数据切分与整合中可能存在的问题

这里，大家应该对数据切分与整合的实施有了一定的认识了，或许很多读者朋友都已经根据各种解决方案各自特性的优劣基本选定了适合于自己应用场景的方案，后面的工作主要就是实施准备了。

在实施数据切分方案之前，有些可能存在的问题我们还是需要做一些分析的。一般来说，我们可能遇到的问题主要会有以下几点：

- 引入分布式事务的问题；
- 跨节点 Join 的问题；
- 跨节点合并排序分页问题；

1. 引入分布式事务的问题

一旦数据进行切分被分别存放在多个 MySQL Server 中之后，不管我们的切分规则设计的多么的完美（实际上并不存在完美的切分规则），都可能造成之前的某些事务所涉及到的数据已经不在同一个 MySQL Server 中了。

在这样的场景下，如果我们的应用程序仍然按照老的解决方案，那么势必需要引入分布式事务来解决。而在 MySQL 各个版本中，只有从 MySQL 5.0 开始以后的各个版本才开始对分布式事务提供支持，而且目前仅有 Innodb 提供分布式事务支持。不仅如此，即使我们刚好使用了支持分布式事务的 MySQL 版本，同时也是使用的 Innodb 存储引擎，分布式事务本身对于系统资源的消耗就是很大的，性能本身也并不是太高。而且引入分布式事务本身在异常处理方面就会带来较多比较难控制的因素。

怎么办？其实我们可以可以通过一个变通的方法来解决这种问题，首先需要考虑的一件事情就是：是否数据库是唯一一个能够解决事务的地方呢？其实并不是这样的，我们完全可以结合数据库以及应用程序两者来共同解决。各个数据库解决自己身上的事务，然后通过应用程序来控制多个数据库上面的事务。

也就是说，只要我们愿意，完全可以将一个跨多个数据库的分布式事务分拆成多个仅处于单个数据库上面的小事务，并通过应用程序来总控各个小事务。当然，这样作的要求就是

我们的俄应用程序必须要有足够的健壮性，当然也会给应用程序带来一些技术难度。

2.　跨节点 Join 的问题

上面介绍了可能引入分布式事务的问题，现在我们再看看需要跨节点 Join 的问题。数据切分之后，可能会造成有些老的 Join 语句无法继续使用，因为 Join 使用的数据源可能被切分到多个 MySQL Server 中了。

怎么办？这个问题从 MySQL 数据库角度来看，如果非得在数据库端来直接解决的话，恐怕只能通过 MySQL 一种特殊的存储引擎 Federated 来解决了。Federated 存储引擎是 MySQL 解决类似于 Oracle 的 DB Link 之类问题的解决方案。和 Oracle DB Link 的主要区别在于 Federated 会保存一份远端表结构的定义信息在本地。咋一看，Federated 确实是解决跨节点 Join 非常好的解决方案。但是我们还应该清楚一点，那就似乎如果远端的表结构发生了变更，本地的表定义信息是不会跟着发生相应变化的。如果在更新远端表结构的时候并没有更新本地的 Federated 表定义信息，就很可能造成 Query 运行出错，无法得到正确的结果。

对待这类问题，我还是推荐通过应用程序来进行处理，先在驱动表所在的 MySQL Server 中取出相应的驱动结果集，然后根据驱动结果集再到被驱动表所在的 MySQL Server 中取出相应的数据。可能很多读者朋友会认为这样做对性能会产生一定的影响，是的，确实是会对性能有一定的负面影响，但是除了此法，基本上没有太多其他更好的解决办法了。而且，由于数据库通过较好的扩展之后，每台 MySQL Server 的负载就可以得到较好的控制，单纯针对单条 Query 来说，其响应时间可能比不切分之前要提高一些，所以性能方面所带来的负面影响也并不是太大。更何况，类似于这种需要跨节点 Join 的需求也并不是太多，相对于总体性能而言，可能也只是很小一部分而已。所以为了整体性能的考虑，偶尔牺牲那么一点点，其实是值得的，毕竟系统优化本身就是存在很多取舍和平衡的过程。

3.　跨节点合并排序分页问题

一旦进行了数据的水平切分之后，可能就并不仅仅只有跨节点 Join 无法正常运行，有些排序分页的 Query 语句的数据源可能也会被切分到多个节点，这样造成的直接后果就是这些排序分页 Query 无法继续正常运行。其实这和跨节点 Join 是一个道理，数据源存在于多个节点上，要通过一个 Query 来解决，就和跨节点 Join 是一样的操作。同样 Federated 也可以部分解决，当然存在的风险也一样。

还是同样的问题，怎么办？我同样仍然继续建议通过应用程序来解决。

如何解决？解决的思路大体上和跨节点 Join 的解决类似，但是有一点和跨节点 Join 不太一样，Join 很多时候都有一个驱动与被驱动的关系，所以 Join 本身涉及到的多个表之间的数据读取一般都会存在一个顺序关系。但是排序分页就不太一样了，排序分页的数据源基本上可以说是一个表（或者一个结果集），本身并不存在一个顺序关系，所以在从多个数据源取数据的过程是完全可以并行的。这样，排序分页数据的取数效率我们可以做的比跨库 Join 更高，所以带来的性能损失相对的要更小，在有些情况下可能比在原来未进行数据切分的数据库中效率更高了。当然，不论是跨节点 Join 还是跨节点排序分页，都会使我们

的应用服务器消耗更多的资源，尤其是内存资源，因为我们在读取访问以及合并结果集的这个过程需要比原来处理更多的数据。

分析到这里，可能很多读者朋友会发现，上面所有的这些问题，我给出的建议基本上都是通过应用程序来解决。大家可能心里开始犯嘀咕了，是不是因为我是 DBA，所以就很多事情都扔给应用架构师和开发人员了？

其实完全不是这样，首先应用程序由于其特殊性，可以非常容易做到很好的扩展性，但是数据库就不一样，必须借助很多其他的方式才能做到扩展，而且在这个扩展过程中，很难避免带来有些原来在集中式数据库中可以解决但被切分开成一个数据库集群之后就成为一个难题的情况。要想让系统整体得到最大限度的扩展，我们只能让应用程序做更多的事情，来解决数据库集群无法较好解决的问题。

14.7 小结

通过数据切分技术将一个大的 MySQL Server 切分成多个小的 MySQL Server，既解决了写入性能瓶颈问题，同时也再一次提升了整个数据库集群的扩展性。不论是通过垂直切分，还是水平切分，都能够让系统遇到瓶颈的可能性更小。尤其是当我们使用垂直和水平相结合的切分方法之后，理论上将不会再遇到扩展瓶颈了。

第 15 章 可扩展性设计之 Cache 与 Search 的利用

前言：

前面章节部分所分析的可扩展架构方案，基本上都是围绕在数据库自身来进行的，这样是否会使我们在寻求扩展性之路的思维受到“禁锢”，无法更为宽广的发散开来。这一章，我们就将跳出完全依靠数据库自身来改善扩展性的问题，将数据服务扩展性的改善向数据库之外的天地延伸！

15.1 可扩展设计的数据库之外延伸

数据库主要就是为应用程序提供数据存取相应的服务，提高数据库的扩展性，也是为了更好的提供数据存取服务能力，同时包括可靠性，高效性以及易用性。所以，我们最根本的目的就是让数据层的存储服务能力得到更好的扩展性，让我们的投入尽可能的与产出成正比。

我们都明白一点，数据本身肯定都会需要有一个可以持久化地方，但是我们是否有必要让我们的所有冗余数据都进行持久化呢？我想读者朋友们肯定都会觉得没有这个必要，只要保证有至少两份冗余的数据进行持久化就足够了。而另外一些为了提高扩展性而产生的冗余数据，我们完全可以通过一些特别的技术来替代需要持久化的数据库，如内存 Cache，Search 以及磁盘文件 Cache 和 Search 等等。

寻求数据库软件本身之外的 Cache 和 Search 来解决数据本身的扩展性，已经成为目前各个大型互联网站点都在积极尝试的一个非常重要的架构升级。因为这不仅仅能更大程度的在整个应用系统提升数据处理层本身的扩展性，而且还能更大限度的提升性能。

对于这种架构方式，目前已经比较成熟的解决方案主要有基于对象的分布式内存 Cache

解决方案 Memcached，高性能嵌入式数据库编程库 Berkeley DB ，功能强大的全文搜索引擎 Lucene 等等。

当然，在使用成熟的第三方产品的同时，偶尔自行实现一些特定应用场景下的 Cache 和 Search，也未尝不是一件值得尝试的事情，而且对于公司的技术积累来说也是很有意义的一件事情。当然，决定自行开发实现之前进行全面的评估是绝对必要的，不仅仅包括自身技术实力，对应用的商业需求也需要有一定的评估才行。

其实，不论是使用现成的第三方成熟解决方案还是自主研发，都是需要在开发资源方面有一定投入的。首先要想很好的和现有 MySQL 数据库更好的结合，就有多种思路存在。可以在数据库端实现和 Cache 或者 Search 的数据通讯（数据更新），也可以在应用程序端直接实现 Cache 与 Search 的数据更新。数据库和 Cache 与 Search 可以处于整体架构中不同的层次，也可以并存于相同的层次。

下面我分别针对使用第三方成熟解决方案以及自主研发来进行一些针对性的分析和架构思路探讨，希望对各位读者朋友有一定的帮助。

15.2 合理利用第三方 Cache 解决方案

使用较为成熟的第三方解决方案最大的优势就在于在节省自身研发成本的同时，还能够在互联网上面找到较多的文档信息，帮助我们解决一些日常遇到的问题还是非常有帮助的。

目前比较流行的第三方 Cache 解决方案主要有基于对象的分布式内存 Cache 软件 Memcached 和嵌入式数据库编程库 Berkeley DB 这两种。下面我将分别针对这两种解决方案做一个分析和架构探讨。

15.2.1 分布式内存 Cache 软件 Memcached

相信对于很多读者朋友来说，Memcached 并不会太陌生了吧，他现在的流行程度已经比 MySQL 并不会差太多了。Memcached 之所以如此的流行，主要是因为以下几个原因：

- 通信协议简单，API 接口清晰；
- 高效的 Cache 算法，基于 libevent 的事件处理机制，性能卓越；
- 面向对象的特性，对应用开发人员来说非常友好；
- 所有数据都存放于内存中，数据访问高效；
- 软件开源，基于 BSD 开源协议；

对于 Memcached 本身细节，这里我就不涉及太多了，毕竟这不是本书的重点。下面我们重点看看如何通过 Memcached 来帮助我们你提升数据服务（这里如果再使用数据库本身

可能会不太合适了）的扩展性。

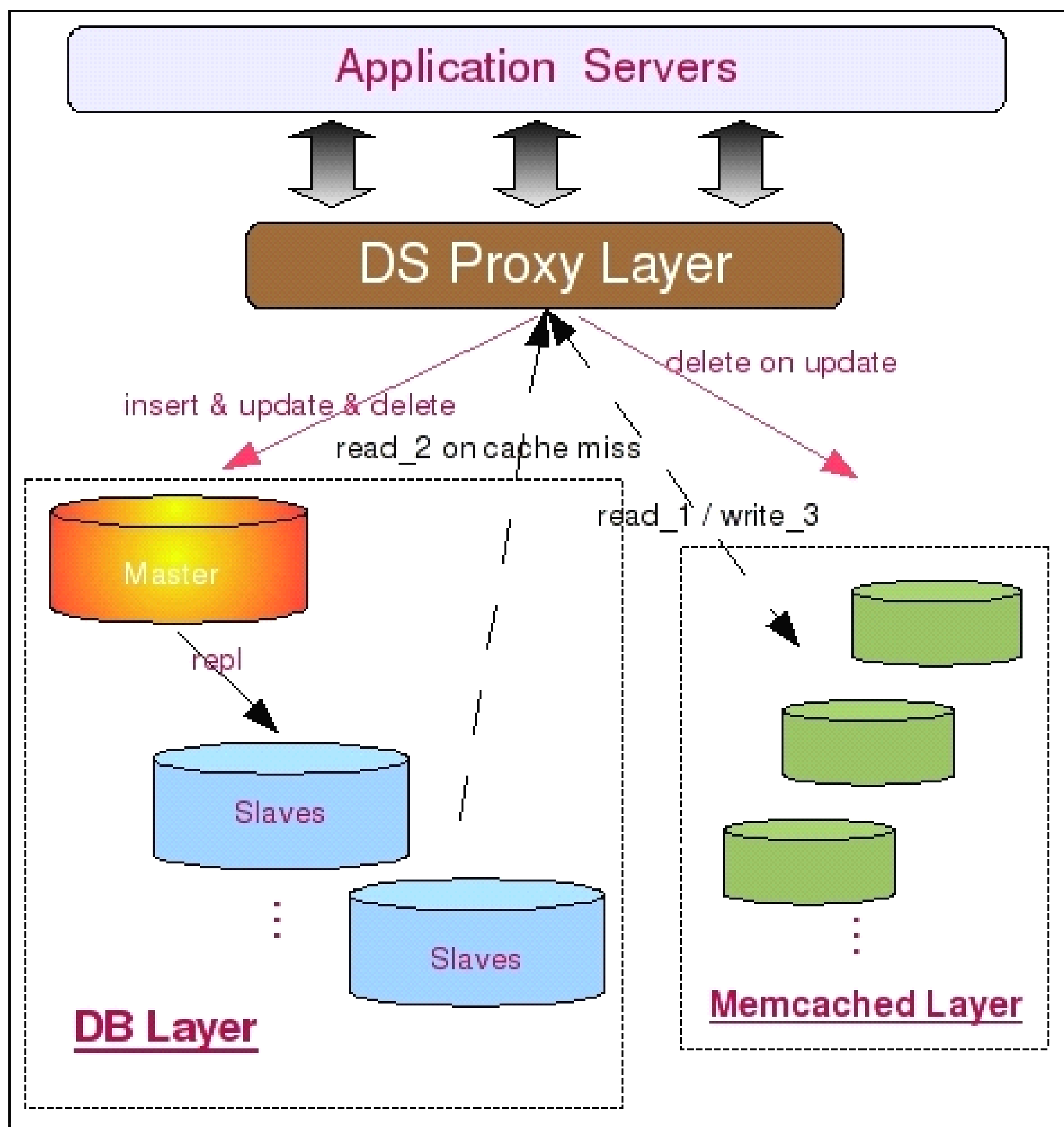
要将 Memcached 较好的整合到系统架构中，首先要在应用系统中让 Memcached 有一个准确的定位。是仅仅作为提升数据服务性能的一个 Cache 工具，还是让他与 MySQL 数据库较好的融合在一起成为一个更为更为高效理想的数据服务层。

1. 作为提升系统性能的 Cache 工具

如果我们仅仅只是系统通过 Memcached 来提升系统性能，作为一个 Cache 软件，那么更多的是需要通过应用程序来维护 Memcached 中的数据与数据库中数据的同步更新。这时候的 Memcached 基本可以理解为比 MySQL 数据库更为前端的一个 Cache 层。

如果我们将 Memcached 作为应用系统的一个数据 Cache 服务，那么对于 MySQL 数据库来说基本上不用做任何改造，仅仅通过应用程序自己来对这个 Cache 进行维护更新。这样作最大的好处就在于可以做到完全不用动数据库相关的架构，但是同时也会有一个弊端，那就是如果需要 Cache 的数据对象较多的时候，应用程序所需要增加的代码量就会增加很多，同时系统复杂度以及维护成本也会直线上升。

下面是将 Memcached 用为简单的 Cache 服务层的时候的架构简图。



从图中我们可以看到，所有数据都会写入 MySQL Master 中，包括数据第一次写入时候的 INSERT，同时也包括对已有数据的 UPDATE 和 DELETE。不过，如果是对已经存在的数据，则需要在 UPDATE 或者 DELETE MySQL 中数据的同时，删除 Memcached 中的数据，以此保证整体数据的一致性。而所有的读请求首先会发往 Memcached 中，如果读取到数据则直接返回，如果没有读取到数据，则再到 MySQL Slaves 中读取数据，并将读取得到的数据写入到 Memcached 中进行 Cache。

这种使用方式一般来说比较适用于需要缓存对象类型少，而需要缓存的数据量又比较大的环境，是一个快速有效的完全针对性能问题的解决方案。由于这种架构方式和 MySQL 数据库本身并没有太大关系，所以这里就不涉及太多的技术细节了。

2. 和 MySQL 整合为数据服务层

除了将 Memcached 用作快速提升效率的工具之外，我们其实还可以将之利用到提高数

据服务层的扩展性方面，和我们的数据库整合成一个整体，或者作为数据库的一个缓冲。

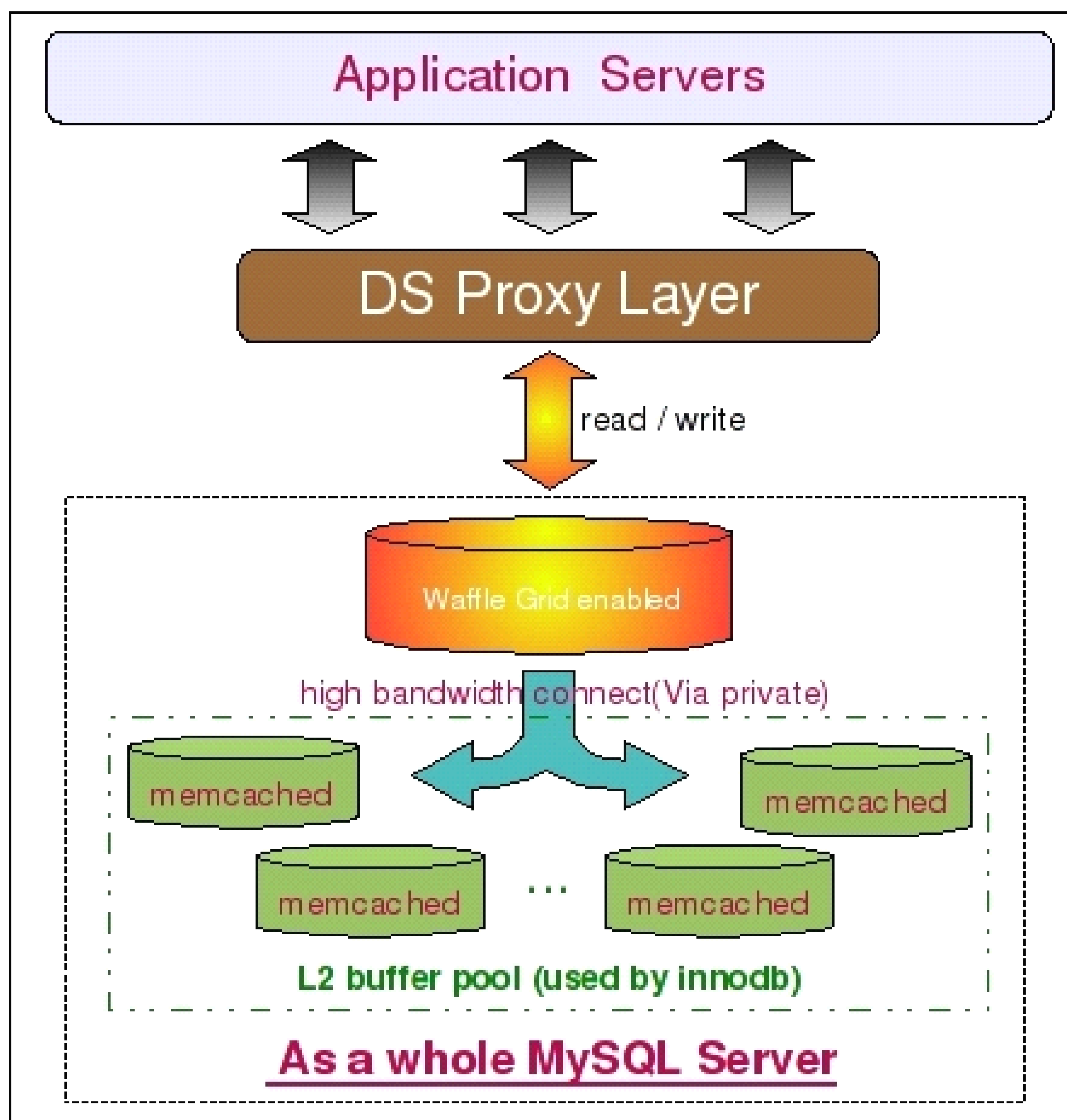
我们首先看看如何将 Memcached 和 MySQL 数据库整合成一个整体来对外提供服务吧。一般来说，我们有两种方式将 Memcached 和 MySQL 数据库整合成一个整体来对外提供数据服务。一种是直接利用 Memcached 的内存容量作为 MySQL 数据库的二级缓存，提升 MySQL Server 的缓存大小，另一种是通过 MySQL 的 UDF 来和 Memcached 进行数据通信，维护和更新 Memcached 中的数据，而应用端则直接通过 Memcached 来读取数据。

对于第一种方式，主要用于业务要求非常特殊，实在难以进行数据切分，而且有很难通过对应用程序进行改造利用上数据库之外的 Cache 的场景。

当然，在正常情况下是肯定无法做到这一点的，之少目前必须借助外界的力量，开源项目 Waffle Grid 就是我们需要借助的外部力量。I

Waffle Grid 是国外的几位 DBA 在工作之余突发奇想出来的一个点子：既然 PC Server 的低廉成本如此的吸引我们，而其 Scale Up 的能力又很难有一个较大的突破，何不利用上现在非常流行的 Memcached 作为突破单台 PC Server 的内存上限呢？就在这个想法的推动下，几位小伙子启动了 Waffle Grid 这个开源项目，利用 MySQL 和 Memcached 双双开源的特性，结合 Memcached 通信协议简单的特点，将 Memcached 成功实现成为 MySQL 主机的外部“二级缓存"，目前仅支持用于 Innodb 的 Buffer Pool。

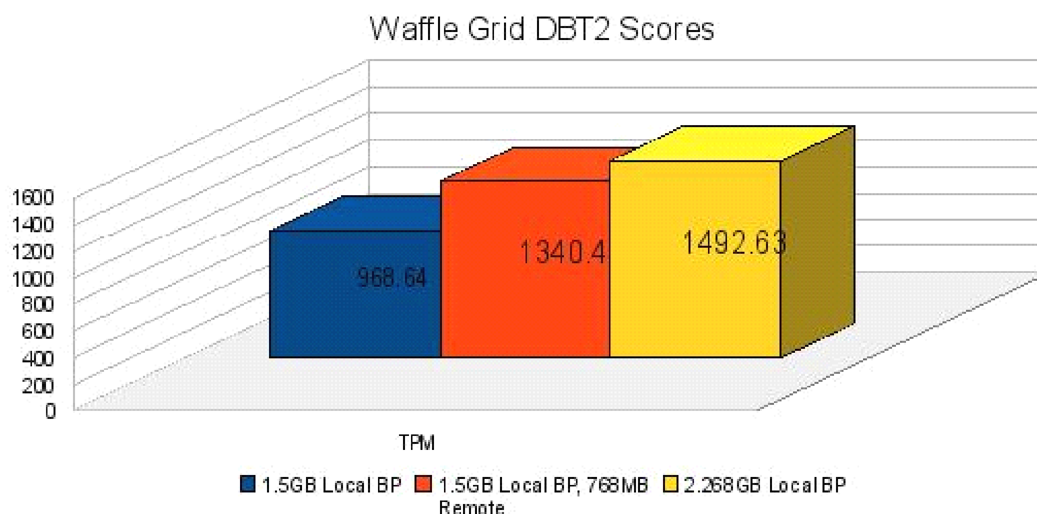
Waffle Grid 的实现原理其实并不复杂，他所做的事情就是当 Innodb 在本地的 Buffer Pool（我们姑且称其为 Local Buffer Pool 吧）的时候，在从磁盘数据文件读取数据之前，先通过 Memcached 的通信 API 接口尝试从 Memcached 中读取相应的缓存数据(我们称之为 Remote Buffer 吧)，只有在 Remote Buffer 中也不存在需要的数据的时候，Innodb 才会访问磁盘文件来读取数据。而且，只有处于 Innodb Buffer pool 中的 LRU List 中的数据会被发送到 Remote Buffer Pool 中，而这些数据一旦被修改，就会 Innodb 就会将之移入 FLUSH List ，Waffle Grid 同时会将进入 FLUSH List 的数据从 Remote Buffer Pool 中清除掉。所以可以说，Remote Buffer Pool 中永远不会存在 Dirty Pages，这也保证了当 Remote Buffer Pool 出现故障的时候不会产生数据丢失的问题。下图是使用 Waffle Grid 项目时候的架构简图：



如架构图上所示，我们首先在 MySQL 数据库端应用 Waffle Grid Patch，通过他连与其他的 Memcached 服务器通信。为了保证网络通信的性能，MySQL 与 Memcached 之间尽可能用高带宽私有网络。

另外，这里的架构图中并没有再将数据库区分 Master 和 Slave 了，并不是说一定不能区分，只是一个示意图。在实际应用过程中，大部分时候只需要在 Slave 上面应用 Waffle Grid 即可，Master 本身并不需要如此大的内存。

看了 Waffle Grid 的实现原理，可能有些读者朋友会有些疑问了。这样做不是所有需要产生物理读的 Query 的性能就会受到直接影响了吗？所有读取 Remote Buffer 的操作都需要通过网络来获取，其性能是否足够高呢？对此，我同样使用作者对 Waffle 的实测数据来接触大家的疑虑：

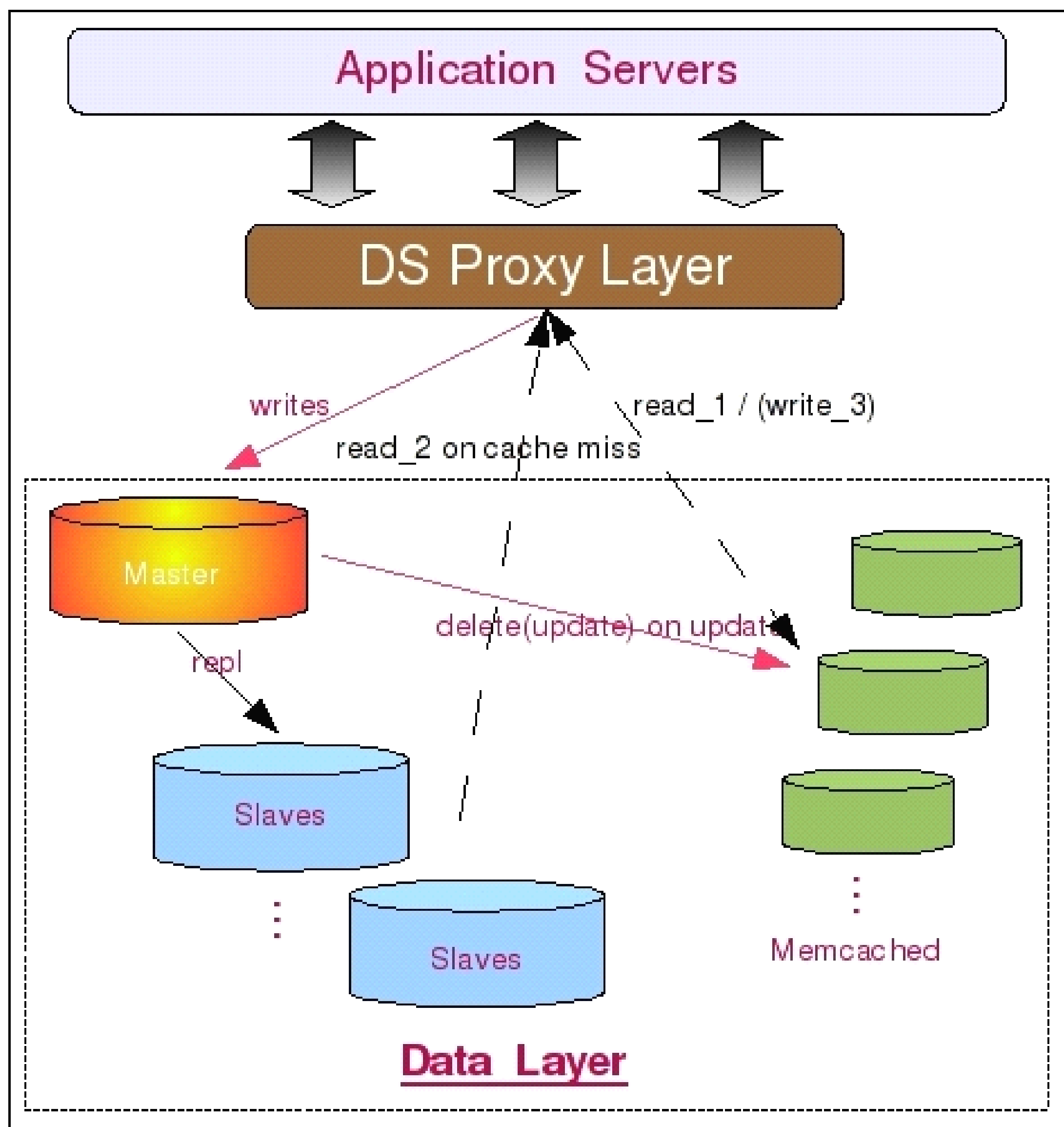


通过 DBT2 所得到的这组测试对比数据，在性能我想并不需要太多的担忧了吧。至于 Waffle Grid 是否适合您的应用场景，那就只能依靠各位读者朋友自己进行评估了。

下面我们再来介绍一下 Memcached 和 MySQL 的另外一种整合方式，也就是通过 MySQL 所提供的 UDF 功能，自行编写相应的程序来实现 MySQL 与 Memcached 的数据通信更新操作。

这种方式和 Waffle Grid 不一样的是 Memcached 中的数据并不完全由 MySQL 来控制维护，而是由应用程序和 MySQL 一起来维护数据。每次应用程序从 Memcached 读取数据的时候，如果发现找不到自己需要的数据，则再转为从数据库中读取数据，然后将读取到的数据写入 Memcached 中。而 MySQL 则控制 Memcached 中数据的失效清理工作，每次数据库中有数据被更新或者被删除的时候，MySQL 则通过用户自行编写的 UDF 来调用 Memcached 的 API 来通知 Memcached 某些数据已经失效并删除该数据。

基于上面的实现原理，我们可以设计出如下这样的一个数据服务层架构：



如图中所示，此架构和上面将 Memcached 完全和 MySQL 读离开作为常规的 Cache 服务器来比较，最大的区别在于 Memcached 的数据变为由 MySQL 数据库来维护更新，而不是应用程序来更新。首先数据被应用程序写入 MySQL 数据库，这时候将会触发 MySQL 上面用户自行编写的相关 UDF，然后通过该 UDF 调用 Memcached 的相关通信接口，将数据写入 Memcached。而当 MySQL 中的数据被更新或者删除的时候，MySQL 中的相关 UDF 同样会更新或者删除 Memcached 中的数据。当然，我们也可以让 MySQL 做更少一些的事情，仅仅只是遇到数据被更新或者删除的时候，通过 UDF 来删除 Memcached 中的数据，写入工作则像前面的架构一样由应用程序来作。

由于 Memcached 基于对象的数据存取，以及通过 Hash 进行数据检索的特性，所以所有存储在 Memcached 中的数据都需要我们设定一个用于标识该数据的 Key，所有数据的存取操作都通过该 Key 来进行。也就是说，如果您并不能像 MySQL 的 Query 语句一样通过某一个（或者多个）关键字条件来读取包含多条数据的结果集，仅适用于通过某个唯一键来获取单条数据的数据读取方式。

15.2.2 嵌入式数据库编程库 Berkeley DB

说实话，数据库编程库这个叫法实在有些别扭，但我也实在找不到其他合适的名词来称呼 Berkeley DB 了，那就姑且使用网上较为通用的叫法吧。

Memcached 所实现的是内存式 Cache，如果我们对性能的要求并没有如此之高，在预算方面也不是太充裕的话，我们还可以选择 Berkeley DB 这样的数据库型 Cache 软件。可能很多读者朋友又会产生疑惑了，我们使用的 MySQL 数据库，为什么还要再使用一个 Berkeley DB 这样的"数据库"呢？实际上 Berkeley DB 在之前也是 MySQL 的存储引擎之一，只不过后期不知道是何原因（获取与商业竞争有关吧），被 MySQL 从支持的存储引擎中移除了。之所以在使用数据库的同时还使用 Berkeley DB 这样的数据库型 Cache，是因为我们可以充分发挥出二者各自的优势，在使用传统通用型数据库的同时，同时可以利用 Berkeley DB 高效的键值对存储方式作为高效数据检索的性能补充，以得到更好的数据服务层扩展性和更高的整体性能。

Berkeley DB 自身架构可以分为五个功能模块，五个模块的在整个系统中相对比较独立，而且可以设置使用或者禁用某一个（或者几个）模块，所以可能称之为五个子系统会更为恰当一些。这五个子系统及基本介绍分别如下：

- 数据存取

  数据存取子系统主要负责最主要也是最基本的数据存与取的工作。而且 Berkeley DB 同时支持了以下四种数据的存储结果方式：Hash，B-Tree，Fixed Length 以及 Dynamic Length。实际上，这四种方式对应了四种数据文件存储的实际格式。数据存储子系统可以完全单独使用，也是必须开启的一个子系统。

- 事务管理

  事务管理子系统主要是针对有事务要求的数据处理服务，提供完整的 ACID 事务属性。在开启事务管理子系统的时候，出了需要开启最基本的数据存取子系统外，还至少需要开启锁管理子系统和日志系统来帮助实现事务的一致性和完整性。

- 锁管理

  锁管理系统主要就是为了保证数据的一致性而提供的共享数据控制功能。支持行级别和页级别的锁定机制，同时为事务管理子系统提供服务。

- 共享内存

  共享内存子系统我想大家看到名称就应该基本知道是做什么事情的了，就是用来管理维护共享 Cache 和 Buffer 的，为系统提升性能而提供数据缓存服务。

- 日志系统

  日志系统主要服务于事务管理系统，为保证事务的一致性，Berkeley DB 也采用先写日志再写数据的策略，一般也都是与事务管理系统同时使用同时关闭。

基于 Berkeley DB 的特性，我们很难像使用 Memcached 那样将他和 MySQL 数据库结合的那么紧密。数据的维护与更新操作主要还是需要通过应用程序来完成。一般来说，在使用 MySQL 的同时还要使用 Berkeley DB 的主要原因就是为了提升系统的性能及扩展性。所以，大多数时候都主要是使用 Hash 和 B-Tree 这两种结构的数据存储格式，尤其是 Hash 格式，是使用最为广泛的，因为这种方式也是存取效率最高的。

在应用程序中，每次数据请求，都先通过预先设定的 Key 到 Berkeley DB 中取查找一次，如果存在数据，则返回取得的数据，如果位检索到数据，则再次到数据库中读取。然后将读取到的数据按照预先设定的 Key，整条存入 Berkeley DB 中，再返回给客户端。而当发生数据修改的时候，应用程序在修改 MySQL 中的数据之后必须还要将 Berkeley DB 中的数据删除。当然，如果您愿意，也可以直接修改 Berkeley DB 中的数据，但是这样就可能引入更多的数据一致性风险并提高系统复杂度了。

从原理来看，使用 Berkeley DB 的方式和将 Memcached 作为纯 Cache 来使用差别不大嘛，为什么我们不用 Memcached 来做呢？其实主要有两个原因，一个是 Memcached 是使用纯内存来存放数据的，而 Berkeley DB 则可以使用物理磁盘，两者在成本方面还是有较大差别的。另外一个原因就是 Berkeley DB 所能支持的数据存储方式除了 Memcached 所使用的 Hash 存储格式之外，同时还可以使用其他存储格式，如 B-Tree 等。

由于和 Memcached 的基本使用原理区别不大，所以这里就不再画图示意了。

15.3 自行实现 Cache 服务

实际上，除了使用比较成熟的现成第三方软件的解决方案之外，如果有一定的技术实力，我们还可以通过自行实现的 Cache 软件来达到完全相同的效果。

当然，您也不要被上面所说的“技术实力”所吓倒，其实也并没有想象中的那么难。只要您不要一开始就希望作出一个能够解决所有问题，而且包含所有其他第三方 Cache 软件的所有优点，还不能遗留任何缺点的软件，不要一开始就希望作出一个多么完美的产品花的软件。从小做起，从精做起。千万别希望一口气吃成一个胖子，这样的解决很可能就是被咽死。

自主研发实现 Cache 服务软件的前提是系统中存在比较特殊的应用场景，通过自主研发可以最大限度的实现比较个性化的需求。当然，也可以针对自己的应用场景进行特定的优化方式来最大限度的提升扩展性和性能。毕竟，只有我们自己才是真正最了解我们的应用系统的人。

决策是否需要自行开发最需要考虑的一个问题就是我的英勇系统场景是否真的如此特别，以至于现成的第三方软件很难解决目前的主要问题？

如果目前的第三方软件已经基本解决了我们系统当前遇到的 80%以上的问题，可能就需要考虑是否有必要完全自主研发了。毕竟我们选择的所有第三方软件都是开源的，如果有某

些小地方无法满足要求，我们完全可以在第三方软件的基础上增加一些我们自己的东西，来满足一些个性化需求。

当我们选择自主研发 Cache 服务软件之后，有以下几点内容是需要注意的：

1. 功能需求
   a) 是完全内存还是可以部分磁盘？
   b) 需要实时同步更新还是可以允许 Cache 数据有延时？
   c) 是否需要支持分布式？

这里所说的功能，实际上就是需求范围的设定。在开始研发之前，我们比需要有一个非常清晰的需求范围，而不是天马行空的边开发边调整，想到啥做啥。毕竟任何软件系统，都是需要以第一线的需求为导向，而且一旦开始开发之后，需求的控制也不能马虎。要不然，很可能就会中途夭折，以失败而告终。

2. 技术实现
   a) 数据同步（或异步）更新机制；
   b) 数据存储方式（Hash Or B-Tree）；
   c) 通讯协议；

技术实现可能会成为研发过程中很大的一个难点，能否有稳定可靠的数据同步（或异步）更新机制决定了该 Cache 软件最终的成败。当然，你可以说数据同步（或异步）更新完全交由需要访问数据的应用程序来自行维护，但是你是否有足够的能力申请在自行研发实现出一个 Cache 软件的同时，还需要前端应用程序作出巨大的调整来适应这个 Cache 软件是一个很大的未知数。老板很可能会说，既然你都自行研发实现了，为啥不能完成数据更新维护功能呢？而数据存储方式直接决定数据的访问方式，同时实现算法也直接决定了软件的性能。最后，数据传输的通讯协议可能也会让人伤透脑筋。如何设计一个足够简单，但是又做到尽可能不会限制后期的扩展升级的通讯协议，可能并不是一件太轻松的事情。毕竟，如果每次升级都需要动到数据传输通讯协议，那每次升级所带来的应用改造成本也太大了。而太过复杂呢，很可能又会影响到前端应用使用的便利性，而且对性能可能也会有一定影响。

3. 可维护性
   a) 方便的管理接口；
   b) 高可用支持（自动或人工切换）；
   c) 基本监控接口；

千万不要忽视了软件系统的维护成本，一个软件一旦开始使用之后，主要工作就是对其进行各种维护。如果可维护性太差，很可能带来极大的维护工作量，甚至带来一线应用人员和运维人员对该软件的信任和使用热情。

使用自行研发的 Cache 服务基于不同的功能特性，可能会有不同的架构组成，但基本上和上面使用 Memcached 所使用的架构区别不大了，所以这里也就不再详细讨论了。

最后，我个人有一个建议就是，在使用比较通用 Cache 服务（也包括自行实现的 Cache 软件服务）的时候，我们应该尽可能将该 Cache 软件与我们的 MySQL 数据库 进行一定的

整合，让彼此能够互补。而且前端的应用程序尽量不要直接操作后端的数据服务集群，尽量通过一个中间代理层来接受处理所有的数据处理服务，对前端应用透明化。这样才能够尽可能做到后端数据服务（数据库与 Cache）层在进行任何扩展的时候，影响到的仅仅只是中间代理层，而对前端完全透明，让我们的数据层拥有真正的高扩展性。

15.4 利用 Search 实现高效的全文检索

不论是使用 Memcached 还是使用 Berkeley DB ，大多数时候都只能通过特定的方式来进行数据的检索，只能满足少部分的检索需求。而数据库本身对于全模糊 LIKE 操作的性能大家应该也很清楚，是非常低下的，因为这种操作无法利用索引。虽然 MySQL 的 MyISAM 存储引擎支持了全文索引，而且官方版本还不支持多字节字符集的数据，所以对于需要存放中文或者需要使用 MyISAM 之外的存储引擎的用户来说，是完全无法使用的。

对于这种情况，我们只有一个办法可以解决，那就是通过全文索引软件，也就是我们常说的 Search（搜索引擎）对数据进行全文索引，才能达到较为高效的数据检索效率。

同样，Search 软件的使用也有使用较为成熟的第三方解决方案与自行研发两种方式。目前最为有名的第三方解决方案主要就是基于 Java 实现的 Lucene，隶属于 Apache 软件基金 Jakarta 项目组下面的一个子项目。当然，他并不是一个完整的搜索引擎工具，而是一个全文检索引擎的框架，他同时提供了完整的用于检索的查询引擎和数据索引引擎。

这里我就不深入讨论 Lucene 本身的技术细节了，感兴趣的读者朋友可以通过访问官方站点（http://lucene.apache.org） 来了解更多也更为权威的细节。我这里主要是介绍一下 Luence 能够给我们带来什么，我们可以怎样来使用他。

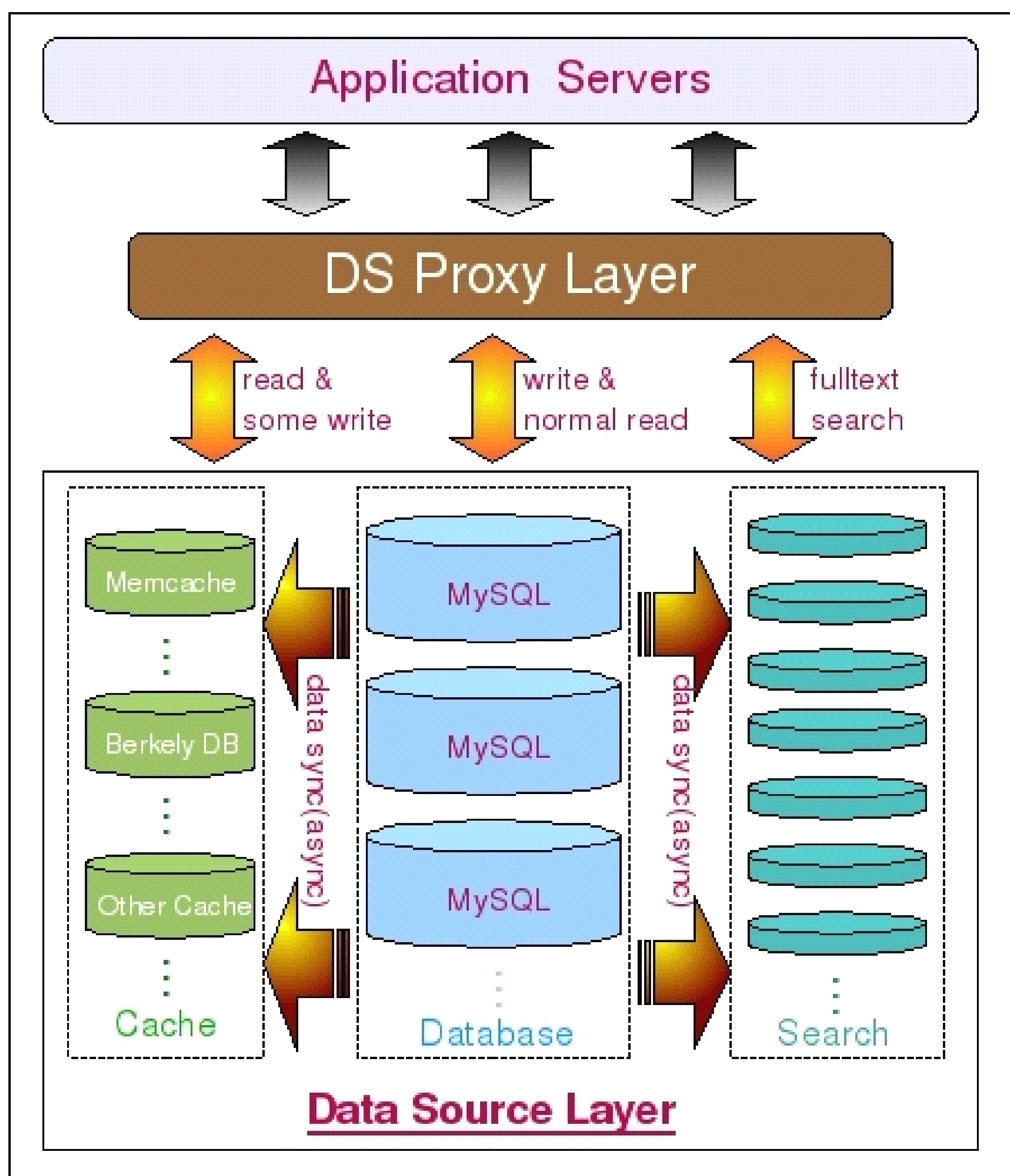
由于 Lucene 高效的全文索引和分词算法，以及高效的数据检索实现，我们完全可以很好的利用这一优点来解决数据库和传统的 Cache 软件完全无法解决的全文模糊搜索功能。我们的需求和传统的通用全网搜索引擎并不一样，并不需要“Spider”到处去爬取互联网上面的数据，只需要将我们数据库中被持久化下来的数据通过应用程序调用 Lucene 的相关 API 写入，并利用 Lucene 创建好索引，然后就可以通过调用 Lucene 所提供的数据检索 API 得到需要访问的数据，而且可以进行全模糊匹配。由于从数据库到 Lucene 这一过程完全由我们自己来实现，所以我们非常容易控制数据的实时性。可以做到完全实时，同样也可以做到固定（或动态）时间段刷新。

虽然 Lucene 的数据也是存放在磁盘上而不是内存中，但是由于高效的分词算法和索引结构，其效率也是非常的好。看到很多网友在网上讨论，当数据量稍微大一些如几十个 G 之后 Lucene 的效率会下降的非常快，其实这是不科学的说法，就从我亲眼所见的场景中，就有好几百 G 的数据在 Lucene 中，性能仍然很出色。这几年性能优化的工作经历及经验中我有一个很深的体会，那就是一个软件性能的好坏，实际上并不仅仅只由其本身所决定，很多时候一个非常高效的软件不同的人使用会有截然不同效果。所以，很多时候当我们使用的第三方软件性能出现问题的时候，不要急着下结论认为是这个软件的问题，更多的是先从自身找找看我们是否真的正确使用了他。

除了使用第三方的 Search 软件如 Lucene 之外，我们也可以自行研发更适用于我们自身应用场景的 Search 软件。就像我目前所供职的公司一样，自行研发了一套纯内存存储的更适合于自身应用场景的高性能分布式 Search 软件，让各个应用系统能够作出很多高效的更为个性化的特色功能。通过多年的技术和经验的积累，现在都已经发展成为和数据库并列的另一个应用系统数据源了。

当然，自行研发 Search 软件的技术门槛可能也比较高，有此技术实力的开发团队并不是很多，所以在决定自行研发之前，一定要做好各方面的评估。不过，如果我们无法实现一个很通用的 Search 软件，但是仅仅只是针对某些特定功能来说，可能实现也并没有想象的那么复杂，更何况如今的开源世界里各种各样的软件数不胜数，利用现有的工具，加上自身个性化定制的二次开发，对于有些特定功能的实现可能就会比较轻松了。

加入了 Search 软件来实现高效的全文检索功能之后，我们的架构可以通过如下这张图来展示：



15.5 利用分布式并行计算实现大数据量的高性能运算

说到大规模大数据量的高性能运算的时候，可能很多人都会想到最近风靡整个 IT 界的一个关键词：云计算，亦或是几年前的“网格计算”。

曾经有朋友建议我将本节标题中的“分布式并行计算”更改为“云计算"，考虑再三之后还是没有更改。说实话，从我个人的理解，不论是“云计算”还是“网格计算"，其实其实质都是一样，都是“分布式并行计算"，只不过是各个商业公司为了吸引大家的眼球所玩的一些“概念游戏”而已，个人认为纯属商业行为。当然，可能有人会认为从“分布式并行计算”到“网格计算”再到“云计算"，每一次“升级”都是在可以利用的计算资源上面有

所扩展。可这种扩展都是在概念上面的扩展，而真正技术实现方面所依靠的并不是这些概念，而是各种软硬件的发展。更何况，最初的“分布式并行计算”概念中本就没有限定我们只能以哪种方式使用哪些资源。

目前比较流行的分布式并行计算框架主要就是以 Google 的 MapReduce 和 Yahoo 的 Hadoop 二者。其实更为准确的说应该是 Google 的 MapReduce + GFS + BigTable 以及 Yahoo 的 Hadoop + HDFS + HBase 这两大架构体系。二者都是由三个负责不同功能的组件组成，MapReduce 与 Hadoop 同为解决认任务分解与合并的功能，GFS 与 HDFS 都是分布式文件系统，解决数据存储的基础设施问题，最后 BigTable 与 HBase 则同为处理结构化数据存储格式的类数据库模块。三大模块共同协作，最终组成一个分布式并行计算的框架体系整体。

其实这分属于两家互联网巨头的分布式并行计算架构框架体系的实现原理基本上可以说是完全一样的。通过前面端的任务分解合并引擎将计算（或者数据存取）任务分解成多个任务，同时发送给多台计算（或数据）节点来进行计算，而后面的每一个节点利用分布式文件系统来作为存储计算数据的基础平台，当然，不论是计算前还是计算后的数据，都是通过 BigTable 或者是 Hbase 这样的模块进行组织。三者组成一个完整的整体，相互依赖。当然，不得不说的是 Hadoop 本身也是由 Google 最初的一篇关于 MapReduce 的论文原理为思想所开发出来的。只不过 Google 在 Open 思想方面所做的贡献很多时候仅限于论文形式，对于其自身技术架构方面的信息公开的实在是有些少。

其实除了这两个重量级的分布式计算框架之外，完全利用现有开源数据库实现的完整解决方案也有一些，如 Inforbright 与 MySQL 合作实现的 BI 解决方案，Greenplum 公司与 Sun 利用 PostGresql 开源数据库实现的 Greenplum 系统，而且两个系统都是依赖 MapReduce 理论所实现。

虽然这两个系统目前并没有前面两个分布式计算框架那样大的伸缩性，但是所针对的场景是实实在在的 DB 场景，数据访问接口完全实现了 SQL 规范。对于使用习惯了通过 SQL 语句来玩数据库的分析方法来说，无疑是非常有诱惑力的，更何况这两个系统基本上都不怎么需要开发，已经是一个完整的产品了。

考虑到篇幅以及非本书重点所在，这里就不深入讨论相关的技术细节了，大家如果对这些内容比较感兴趣，可以通过各自官方网站了解更多更为全面的信息。

15.6 小结

数据库只是存储数据的一种工具，其特殊性只是能将数据持久化，且提供统一规范的访问接口而已。除了数据库，其实我们还可以很多其他的数据存储处理方式，结合各种数据存储处理方式，充分发挥各自的特性，扬长避短，形成一个综合的数据中心，这样才能让系统的数据处理系统的扩展性得到最大的提升，性能得到最优化。

第 16 章 MySQL Cluster

前言：

第 16 章 MySQL Cluster

前言：

MySQL Cluster 是一个基于 NDB Cluster 存储引擎的完整的分布式数据库系统。不仅仅具有高可用性，而且可以自动切分数据，冗余数据等高级功能。和 Oracle Real Cluster Application 不太一样的是，MySQL Cluster 是一个 Share Nothing 的架构，各个 MySQL Server 之间并不共享任何数据，高度可扩展以及高度可用方面的突出表现是其最大的特色。虽然目前还只是 MySQL 家族中的一个新兴产品，但是已经有不少企业正在积极的尝试使用了。本章我们将通过对 MySQL Cluster 的了解来寻找其在可扩展设计方面的优势。

16.1 MySQL Cluster 介绍

简单的说，MySQL Cluster 实际上是在无共享存储设备的情况下实现的一种完全分布式数据库系统，其主要通过 NDB Cluster（简称 NDB）存储引擎来实现。MySQL Cluster 刚刚诞生的时候可以说是一个可以对数据进行持久化的内存数据库，所有数据和索引都必须装载在内存中才能够正常运行，但是最新的 MySQL Cluster 版本已经可以做到仅仅将所有索引装载在内存中即可，实际的数据可以不用全部装载到内存中。

一个 MySQL Cluster 的环境主要由以下三部分组成：

a) SQL 层的 SQL 服务器节点(后面简称为 SQL 节点)，也就是我们常说的 MySQL Server。

主要负责实现一个数据库在存储层之上的所有事情，比如连接管理，Query 优化和响应，Cache 管理等等，只有存储层的工作交给了 NDB 数据节点去处理了。也就是说，在纯粹的 MySQL Cluster 环境中的 SQL 节点，可以被认为是一个不需要提供任何存储引擎的 MySQL 服务器，因为他的存储引擎有 Cluster 环境中的 NDB 节点来担任。所以，SQL 层各 MySQL 服务器的启动与普通的 MySQL Server 启动也有一定的区别，必须要添加 ndbcluster 参数选项才行。我们可以添加在 my.cnf 配置文件中，也可以通过启动命令行来指定。

b) Storage 层的 NDB 数据节点，也就是上面说的 NDB Cluster。

最初的 NDB 是一个内存式存储引擎，当然也会将数据持久化到存储设备上。但是最新的 NDB Cluster 存储引擎已经改进了这一点，可以选择数据是全部加载到内存中还是仅仅加载索引数据。NDB 节点主要是实现底层数据存储功能，来保存 Cluster 的数据。每一个 Cluster 节点保存完整数据的一个 fragment，也就是一个数据分片（或者一份完整的数据，视节点数目和配置而定），所以只要配置得当，MySQL Cluster 在存储层不会出现单点的问题。一般来说，NDB 节点被组织成一个一个的 NDB Group，一个 NDB Group 实际上就是一组存有完全相同的物理数据的 NDB 节点群。

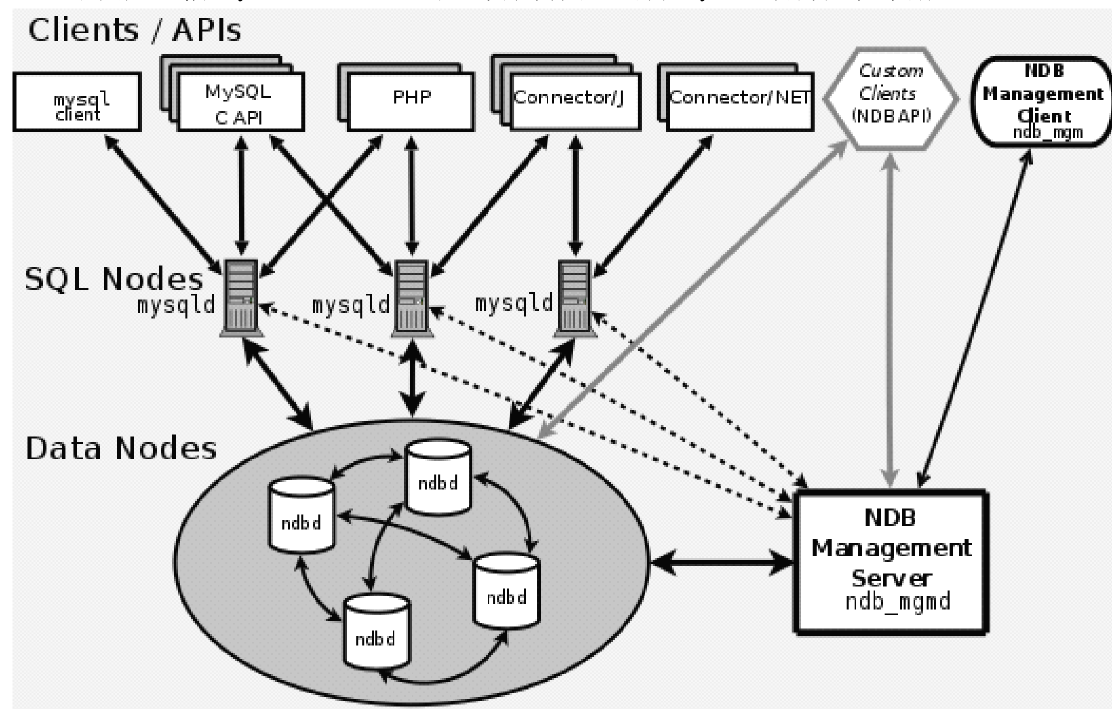
上面提到了 NDB 各个节点对数据的组织，可能每个节点都存有全部的数据也可能只保存一部分数据，主要是受节点数目和参数来控制的。首先在 MySQL Cluster 主配置文件（在管理节点上面，一般为 config.ini）中，有一个非常重要的参数叫 NoOfReplicas，这个参数指定了每一份数据被冗余存储在不同节点上面的份数，该参数一般至少应该被设置成 2，也只需要设置成 2 就可以了。因为正常来说，两个互为冗余的节点同时出现故障的概率还是非常小的，当然如果机器和内存足够多的话，也可以继续增大来更进一步减小出现故障的概率。此外，一个节点上面是保存所有的数据还是一部分数据还受到存储节点数目的限制。NDB 存储引擎首先保证 NoOfReplicas 参数配置的要求来使用存储节点，对数据进行冗余，然后再

根据节点数目将数据分段来继续使用多余的 NDB 节点。分段的数目为节点总数除以 NoOfReplicas 所得。

c）负责管理各个节点的 Manage 节点主机：

管理节点负责整个 Cluster 集群中各个节点的管理工作，包括集群的配置，启动关闭各节点，对各个节点进行常规维护，以及实施数据的备份恢复等。管理节点会获取整个 Cluster 环境中各节点的状态和错误信息，并且将各 Cluster 集群中各个节点的信息反馈给整个集群中其他的所有节点。由于管理节点上保存了整个 Cluster 环境的配置，同时担任了集群中各节点的基本沟通工作，所以他必须是最先被启动的节点。

下面是一幅 MySQL Cluster 的基本架构图（出自 MySQL 官方文档手册）：



通过图中我们可以更清晰的了解整个 MySQL Cluster 环境各个节点以及客户端应用之间的关系。

由于 MySQL Cluster 目前的成熟使用并不是太多，实现也较普通的 MySQL 略复杂，所以本章将首先从如何搭建一个 MySQL Cluster 环境开始来介绍他。

16.2 MySQL Cluster 环境搭建

搭建 MySQL Cluster 首先需要至少一个管理节点主机来实现管理功能，一个 SQL 节点主机来实现 MySQL server 功能和两个 ndb 节点主机实现 NDB Cluster 的功能。在后面的介绍中，我采用双 SQL 节点来搭建测试环境，具体信息如下：

1、硬件准备

a）MySQL 节点 1　　192.168.0.1

b）MySQL 节点 2　　　192.168.0.2
c）ndb 节点 1　　　　192.168.0.3
d）ndb 节点 2　　　　192.168.0.4
e）管理节点　　　　　192.168.0.5

2、软件安装

首先在上面 5 个节点的主机上尽量确保环境基本一致，然后从 MySQL 官方下载相应的软件包并分发到两台 SQL 节点和两台 NDB 节点上，以备后面的安装时候的使用。

我的测试环境 OS（RedHat Linux）如下（非必须）：

root@mysql1:/usr/local>uname -a

Linux oratest1 2.6.9-42.ELsmp #1 SMP Wed Jul 12 23:27:17 EDT 2006 i686 i686 i386 GNU/Linux

a）安装 MySQL 节点：

在 MySQL 节点上面需要安装支持 cluster 的 MySQL Server，可以通过自编译源代码安装也可以选择 MySQL 官方提供的编译好的 tar 包或者 rpm 安装包，我是通过源代码自行编译的，实际上完全可以通过 MySQL 官方提供的经过优化编译的二进制 tar 包，只是我自己习惯了而已，我的编译设置参数如下：

```
root@mysql1>./configure \
--prefix=/usr/local/MySQL \
--without-debug \
--without-bench \
--enable-thread-safe-client \
--enable-assembler \
--with-charset=utf8 \
--with-extra-charsets=complex \
--with-client-ldflags=-all-static \
--with-MySQLd-ldflags=-all-static \
--with-ndbcluster \
--with-server-suffix=-max \
--datadir=/data/mysqldata \
--with-unix-socket-path=/usr/local/MySQL/sock/mysql.sock
...

root@mysql1>make
...

root@mysql1>make install
...
```

然后是配置设置配置文件/etc/my.cnf，由于是测试环境，所以我仅仅设置了 ndbcluster 所需要的最基本的两个配置项，其他所有的配置均用默认配置（后面会有较为详细的配置说明），如下：

root@mysql1>vi /etc/my.cnf
[client]
socket = /usr/local/mysql/sock/mysql.sock #由于编译时候特殊指定了,所以设置在这里，方便以后登入的时候使用
[MySQLd]
socket = /usr/local/mysql/sock/mysql.sock
ndbcluster

[MySQL_cluster]
ndb-connectstring = 192.168.0.5

继续完成后面的 MySQL 安装过程：
root@mysql1>cd /usr/local/mysql
root@mysql1>bin/mysql_install_db --user=mysql --socket=/usr/local/mysql/sock/mysql.sock
Installing MySQL system tables...
OK
Filling help tables...
OK

To start MySQLd at boot time you have to copy
support-files/MySQL.server to the right place for your system

PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:
/usr/local/mysql/bin/MySQLadmin -u root password 'new-password'
/usr/local/mysql/bin/MySQLadmin -u root -h ointest_stb password 'new-password'

Alternatively you can run:
/usr/local/mysql/bin/MySQL_secure_installation

which will also give you the option of removing the test
databases and anonymous user created by default. This is
strongly recommended for production servers.

See the manual for more instructions.

You can start the MySQL daemon with:
cd /usr/local/MySQL ; /usr/local/mysql/bin/MySQLd_safe &

You can test the MySQL daemon with MySQL-test-run.pl
cd MySQL-test ; perl MySQL-test-run.pl

```
Please report any problems with the /usr/local/mysql/bin/MySQLbug script!

The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at http://shop.mysql.com

root@mysql1>chown -R root  .
root@mysql1>chgrp -R mysql .
root@mysql1>chown -R mysql.mysql /usr/local/mysql/etc
root@mysql1>chown -R mysql.mysql /usr/local/mysql/sock
root@mysql1>chown -R mysql.mysql /usr/local/mysql/log
root@mysql1>:/usr/local/mysql# ls -l
total 40
drwxr-xr-x    2 root     MySQL        4096 May  4 14:47 bin
drwxr-xr-x    2 MySQL    MySQL        4096 May  4 14:20 etc
drwxr-xr-x    3 root     MySQL        4096 May  4 14:46 include
drwxr-xr-x    2 root     MySQL        4096 May  4 14:46 info
drwxr-xr-x    3 root     MySQL        4096 May  4 14:46 lib
drwxr-xr-x    2 root     MySQL        4096 May  4 14:47 libexec
drwxr-xr-x    2 MySQL    MySQL        4096 May  4 14:20 log
drwxr-xr-x    4 root     MySQL        4096 May  4 14:47 man
drwxr-xr-x    9 root     MySQL        4096 May  4 14:47 MySQL-test
drwxr-xr-x    2 MySQL    MySQL        4096 May  5 22:16 sock
root@mysql1>:/usr/local/mysql#
```

b） 安装 ndb 节点：

如果希望尽可能的各环境保持一致，建议在 NDB 节点也和 SQL 节点一样安装整个带有 NDB Cluster 存储引擎的 MySQL Server。由于安装细节和上面的 SQL 节点完全一样，所以这里就不再累述。

另外，如果只是为了保证能够完整的 MySQL Cluster 这个环境，则在 NDB 节点上完全可以仅安装 NDB 存储引擎（mysql ndb storage engine）即可。安装 NDB 存储引擎好像目前是找不到源码来自行编译安装的，只能通过 MySQL AB 官方提供的 rpm 包来安装。安装过程非常简单，和其他的 rpm 软件包安装没有任何区别。

c） 管理节点：

管理节点所需要的安装更简单，实际上只需要 ndb_mgm 和 ndb_mgmd 两个程序即可，这两个可执行程序可以在上面的 MySQL 节点的 MySQL 安装目录中的 bin 目录下面找到。将这两个程序 copy 到管理节点上面合适的位置（自行考虑，我一般会放在 /usr/local/mysql/bin 下面），并在 path 制定的目录中建立两个同名的 soft link 在到这两个程序上面，就可以了。

以上即是 MySQL Cluster 环境的软件安装过程，看上去并不复杂是吧，希望大家的安装过程也能够一切顺利，当然如果遇到了什错误也不用担心，MySQL 官方手册中也提供了非常详细的安装过程说明。

3、基本配置

在上面所有节点的软件安装完成之后，就是MySQL Cluster环境的配置工作了。如果不考虑其他一些优化和个性化的配置需求，MySQL Cluster的基本配置是比较简单的。这里暂时先仅仅完成一个简单的测试环境的配置，详细的配置说明请看后面的MySQL Cluster配置介绍的章节。

对于MySQL节点和ndb节点在上面的安装过程中已经完成了，仅需要设置[MySQL_cluster]参数组的ndb-connectstring参数即可完成最基本的配置。

管理节点的配置稍微复杂一点，因为他需要配置出Cluster环境中每一个节点的基本信息。配置文件并不需要一个特别固定的位置和名称，都由用户自行设定，只需要在启动过程中指定配置文件即可。在我们的测试环境中配置为建名称为/var/lib/MySQL-cluster/config.ini，内容如下：

```
[root@mysqlMgm ~]# cat  /var/lib/mysql-cluster/config.ini
[NDBD DEFAULT]
NoOfReplicas=2
DataMemory=64M
IndexMemory=16M

[TCP DEFAULT]
portnumber=2202

#管理节点
[NDB_MGMD]
id=1
hostname=192.168.0.5
datadir=/var/lib/mysql-cluster

#第一个ndbd节点:
[NDBD]
id=2
hostname=192.168.0.3
datadir=/data/mysqldata

#第二个ndbd节点:
[NDBD]
id=3
hostname=192.168.0.4
datadir=/drbddata/mysqldata

# SQL node options:
[MySQLD]
id=4
hostname=192.168.0.1
```

```
[MySQLD]
id=5
hostname=10.0.65.203
[root@mysqlMgm ~]#
```

1）SQL 节点的配置:

MySQL 节点的配置和普通的 MySQL Server 的配置区别主要是需要在 my.cnf 文件中增加[mysql_cluster]这个配置选项组，并至少指定 ndb-connectstring=192.168.0.5，也就是制定管理节点的 ip 地址或者 hostname。另外，如果希望能在启动 MySQLd 的时候不用手动指定 ndbcluster 参数，则在[mysqld]参数选项组中增加 ndbcluster 项参数。除了这两项之外，其他的所有参数都可以可以使用默认值。

2）NDB 存储节点的配置：

NDB 存储节点的配置就更简单的了，仅仅需[mysql_cluster]中的 ndb-connectstring = 192.168.0.5 参数，其他所有的都可以不再配置了。

4、环境测试

在 MySQL Cluster 环境搭建完成后，首先肯定要对新搭建的环境进行一些基本的功能和异常测试，以确认搭建的环境是否已经可以正常提供服务。

1）首先检测 ndb 引擎是否已经正常工作

通过任意客户端连接任意选定的一个 SQL 节点，测试各种基本的 ddl，dml 操作，然后再通过客户端连接上Cluster环境中另外的SQL节点校验所作的草食是否在其他节点同样可见了。下面是测试 create table 后再插入一条数据的示例：

在节点 4 上面：

```
mysql>use test;
mysql>create table t1 ( a int) engine=ndb;
Query ok, 0 rows affected (0.00 sec)
mysql>insert into t1 values(100);
Query ok, 1 rows affected (0.00 sec)
```

然后在节点 5 上面：

```
mysql>use test;
mysql>select * from t1;
+-----+
| id  |
+-----+
| 100 |
+-----+
1 row in set (0.00 sec)
```

可见，在节点 4 上面所插入的数据，已经在节点 5 上面了，说明 ndb 引擎工作正常的。其他的测试与此类似，大家可以自行测试。

如果在测试中发现在某两个节点之间出现不一致现象，那么可以肯定的是，Cluster 环境的配置有问题。在管理节点上面通过“ndb_mgm -e SHOW”命令查看各节点状态是否正

常，是否都已经连接到了管理节点上面。并检查不正常节点的my.cnf配置文件，是否已经配置好了以ndbcluster方式启动MySQLd，是否有正确配置[mysql_cluster]这个参数组的最基本的ndb-connectstring参数。然后检查管理节点上面的config文件，里面是否有正确配置好各所有节点的配置，尤其是不正常的SQL节点的配置。

2）检测冗余环境的单点故障问题

a、模拟NDB节点Crash

由于是模拟Crash，所以我们通过在节点2上面kill掉ndb进程，然后再分别通过两个SQL节点去访问t1表，查看是否可以正常访问，数据是否一样。

在节点4上面：

```
mysql> use test;
mysql> select * from t1;
+------+
| id  |
+------+
| 100 |
+------+
1 row in set (0.00 sec)
mysql> insert into t1 values(200);
Query ok, 1 rows affected (0.00 sec)
```

在节点5上面：

```
mysql>use test;
mysql>select * from t1;
+------+
| id  |
+------+
| 100 |
| 200 |
+------+
2 row in set (0.00 sec)
mysql> delete from t1 where id = 100;
Query ok, 1 rows affected (0.00 sec)
```

再回到节点4上面：

```
mysql> select * from t1;
+------+
| id  |
+------+
| 200 |
+------+
1 row in set (0.00 sec)
```

可以看到，不仅t1仍然可以正常访问，数据也没有任何丢失，且仍然可以正常插入，删除数据。可见，在有一个NDB节点Crash之后，真个MySQL Cluster环境仍然可以正

常提供服务。当然，如果两个NDB节点都Crash之后，MySQL Cluster环境就无法正常提供服务了，大家也可以自行测试一下。

b、模拟SQL节点Crash

同样和测试NDB节点Crash一样，kill掉一个SQL节点（比如节点4）的mysqld进程，然后通过节点5进行访问：

在节点5上面：

```
mysql> use test;
mysql> select * from t1;
+-----+
| id  |
+-----+
| 200 |
+-----+
1 row in set (0.00 sec)
mysql> insert into t1 values(300);
Query ok, 1 rows affected (0.00 sec)
mysql> select * from t1;
+-----+
| id  |
+-----+
| 200 |
| 300 |
+-----+
2 row in set (0.00 sec)
```

可以看到，当节点4 Crash之后，节点5仍然能够提供正常的服务。当然，如果在应用环境中,应用环境需要至少支持当一个SQL节点出现问题的时候能够自行切换到剩下的正常的SQL节点来访问。

c、管理节点的单点

一般情况来说，管理节点是最容易控制的，实施也非常简单，只需要将配置文件和两个可执行程序（ndb_mgmd和ndb_mgm）存放在多台机器上面即可，所以一般来说不需要太多考虑单点故障。

16.3 MySQL Cluster **配置详细介绍**（config.ini）

在MySQL Cluster环境的配置文件config.ini里面，每一类节点都有两个（或以上）的相应配置项组，每一类节点的配置项都主要由两部分组成，一部分是同类所有节点相同的配置项组，在[NDB_MGM DEFAULT]、[NDBD DEFAULT]和[MySQLD DEFAULT]这三个配置组里面,而且每一个配置组只出现一次;而另外一部分则是针对每一个节点独有配置内容的配置项组[NDB_MGM]、[NDBD]和[MySQLD]，由于这三类配置组中配置的每一个节点独有的个性化配置,

所以每一个配置组都可能会出现多次（每一个节点一次）。下面是每一类节点的各种配置说明：

1、管理节点相关配置

在整个 MySQL Cluster 环境中，管理节点相关的配置为[NDBD_MGM DEFAULT]和[NDB_MGMD]相关的两组：

1）[NDB_MGMD DEFAULT]中各管理节点的共用配置项：

PortNumber：配置管理节点的服务端程序（ndb_mgmd）监听客户端（ndb_mgm）连接请求和发送的指令，从文档上可以查找到，默认端口是 1186 端口。一般来说这一项不需要更改，当然如果是为了在同一台主机上面启动多个管理节点的话，肯定需要将两个管理节点启动不同的监听端口；

LogDestination：配置管理节点上面的 cluster 日志处理方式。

a）可以写入文件如：LogDestination=FILE:filename=my-cluster.log,maxsize=500000,maxfiles=4;

b）也可以通过标准输出来打印出来如：LogDestination=CONSOLE;

c）还可以计入 syslog 里面如：LogDestination=SYSLOG:facility=syslog;

d）甚至多种方式共存：LogDestination=CONSOLE;SYSLOG:facility=syslog;FILE:filename=/var/log/cluster-log

Datadir:设置用于管理节点存放文件输出的位置。如 process 文件(.pid),cluster log 文件（当 LogDestination 有 FILE 处理方式存在时候）。

ArbitrationRank：配置各节点在处理某些事件出现分歧的时候的级别。有 0，1,2 三个值可以选择。

a）0 代表本节点完全听其他节点的，不参与决策

b）1 代表本节点有最高优先权，“一切由我来决策”

c）2 代表本节点参与决策，但是优先权较 1 低，但是比 0 高

ArbitrationRank 参数不仅仅管理节点有，MySQL 节点也有。而且一般来说，所有的管理节点一般都应该设置成 1，所有 SQL 节点都设置成 2。

2）[NDB_MGMD]是每个管理节点配置一组，所需配置项如下（下面的参数只能设置在[NDB_MGMD]参数组中）：

Id：为节点指定一个唯一的 ID 号，要求在整个 Cluster 环境中唯一；

Hostname：配置该节点的 IP 地址或者主机名，如果是主机名，则该主机名必须要在配置文件所在的节点的/etc/hosts 文件中存在，而且绑定的 IP 是准确的。

上面[NDB_MGMD DEFAULT]里面的所有参数项，都可以设置在下面的[NDB_MGMD]参数组里面，但是 Id 和 Hostname 两个参数只能设置在[NDB_MGMD]里面，而不能设置在[NDB_MGMD DEFAULT]里面，因为这两个参数项针对每一个节点都是不相同的内容。

2、NDB 节点相关配置

NDB 节点和管理节点一样，既有各个节点共用的配置信息组[NDBD DEFAULT]，也有每一个节点个性化配置的[NDBD]配置组（实际上 SQL 节点也是如此）。

1）[NDBD DEFAULT]中的配置项：

NoOfReplicas：定义在Cluster环境中相同数据的分数，通俗一点来说就是每一份数据存放NoOfReplicas份。如果希望能够冗余，那么至少设置为2（一般情况来说此参数值设置为2就够了），最大只能设置为4。另外，NoOfReplicas值得大小，实际上也就是node group大小的定义。NoOfReplicas参数没有系统默认值，所以必须设定，而且只能设置在[NDBD DEFAULT]中，因为此数值在整个Cluster集群中一个node group中所有的NDBD节点都需要一样。另外NoOfReplicas的数目对整个Cluster环境中NDB节点数量有较大的影响，因为NDB节点总数量是NoOfReplicas * 2 * node_group_num；

DataDir：指定本地的pid文件，trace文件，日志文件以及错误日志子等存放的路径，无系统默认地址，所以必须设定；

DataMemory：设定用于存放数据和主键索引的内存段的大小。这个大小限制了能存放的数据的大小，因为ndb存储引擎需属于内存数据库引擎，需要将所有的数据（包括索引）都load到内存中。这个参数并不是一定需要设定的，但是默认值非常小（80M），只也就是说如果使用默认值，将只能存放很小的数据。参数设置需要带上单位，如512M，2G等。另外，DataMemory里面还会存放UNDO相关的信息，所以，事务的大小和事务并发量也决定了DataMemory的使用量，建议尽量使用小事务；

IndexMemory：设定用于存放索引（非主键）数据的内存段大小。和DataMemory类似，这个参数值的大小同样也会限制该节点能存放的数据的大小，因为索引的大小是随着数据量增长而增长的。参数设置也如DataMemory一样需要单位。IndexMemory默认大小为18M；

实际上，一个NDB节点能存放的数据量是会受到DataMemory和IndexMemory两个参数设置的约束，两者任何一个达到限制数量后，都无法再增加能存储的数据量。如果继续存入数据系统会报错“table is full”。

FileSystemPath：指定redo日志，undo日志，数据文件以及meta数据等的存放位置，默认位置为DataDir的设置，并且在ndbd初始化的时候，参数所设定的文件夹必须存在。在第一次启动的时候，ndbd进程会在所设定的文件夹下建立一个子文件夹叫ndb_id_fs，这里的id为节点的ID值，如节点id为3则文件夹名称为ndb_3_fs。当然，这个参数也不一定非得设置在[NDBD DEFAULT]参数组里面让所有节点的设置都一样(不过建议这样设置)，还可以设置在[NDBD]参数组下为每一个节点单独设置自己的FileSystemPath值；

BackupDataDir：设置备份目录路径，默认为FileSystemPath/BACKUP。

接下来的几个参数也是非常重要的，主要都是与并行事务数和其他一些并行限制有关的参数设置。

MaxNoOfConcurrentTransactions：设置在一个节点上面的最大并行事务数目，默认为4096，一般情况下来说是足够了的。这个参数值所有节点必须设置一样，所以一般都是设置在[NDBD DEFAULT]参数组下面；

MaxNoOfConcurrentOperations：设置同时能够被更新（或者锁定）的记录数量。一般来说可以设置为在整个集群中相同时间内可能被更新(或者锁定)的总记录数，除以NDB节点数，所得到的值。比如，在集群中有两个NDB节点，而希望能够处理同时更新（或锁定）100000条记录，那么此参数应该被设置为：100000 / 4 = 25000。此外，这里的记录数量并不是指单纯的表里面的记录数，而是指事物里面的操作记录。当使用到唯一索引的时候，表的数据和索引两者都要算在里面，也就是说，如果是通过一个唯一索引来作为过滤条件更新某一条记录，那么这里算是两条操作记录。而且即使是锁定也会产生操作记录，比如通过唯一索引来查找一条记录，就会产生如下两条操作记录：通过读取唯一索引中的某个记录数据会产生锁定，产生一条操作记录，然后读取基表里面的数据，这里也会产生读锁，也会产生一条操作记录。MaxNoOfConcurrentOperations参数的默认值为32768。当我们额度系统运行过程中，如果出现此参数不够的时候，就会报出"Out of operation records in transaction coordinator"这样的错误信息；

MaxNoOfLocalOperations：此参数默认是MaxNoOfConcurrentOperations * 1.1的大小，也就是说，每个节点一般可以处理超过平均值的10%的操作记录数量。但是一般来说，MySQL建议单独设置此参数而不要使用默认值，并且将此参数设置得更较大一些；

以下的三个参数主要是在一个事务中执行一条query的时候临时用到存储(或者内存）的情况下所使用到的，所使用的存储信息会在事务结束（commit或者rollback）的时候释放资源；

MaxNoOfConcurrentIndexOperations：这个参数和MaxNoOfConcurrentOperations参数比较类似，只不过所针对的是Index的record而已。其默认值为8192，对伊一般的系统来说都已经足够了，只有在事务并发非常非常大的系统上才有需要增加这个参数的设置。当然，此参数越大，系统运行时候为此而消耗的内存也会越大；

MaxNoOfFiredTriggers：触发唯一索引（hash index）操作的最大的操作数，这个操作数是影响索引的操作条目数，而不是操作的次数。系统默认值为4000，一般系统来说够用了。当然，如果系统并发事务非常高，而且涉及到索引的操作也非常多，自然也就需要提高这个参数值的设置了；

TransactionBufferMemory:这个buffer值得设置主要是指定用于跟踪索引操作而使用的。主要是用来存储索引操作中涉及到的索引key值和column的实际信息。这个参数的值一般来说也很少需要调整，因为实际系统中需要的这部分buffer量非常小，虽然默认值只是1M，但是对于一般应用也已经足够了；

下面要介绍到的参数主要是在系统处理中做table scan或者range scan的时候使用的一些buffer的相关设置，设置的恰当可以既节省内存又达到足够的性能要求。

MaxNoOfConcurrentScans:这个参数主要控制在Cluster环境中并发的table scan和range scan的总数量平均分配到每一个节点后的平均值。一般来说，每一个scan都是通过并行的扫描所有的partition来完成的，每一个partition的扫描都会在该partition

所在的节点上面使用一个 scan record。所以，这个参数值得大小应该是“scan record”数目 * 节点数目。参数默认大小为 256，最大只能设置为 500；

MaxNoOfLocalScans：和上面的这个参数相对应，只不过设置的是在本节点上面的并发 table scan 和 range scan 数量。如果在系统中有大量的并发而且一般都不使用并行的话，需要注意此参数的设置。默认为 MaxNoOfConcurrentScans * node 数目；

BatchSizePerLocalScan：该参用于计算在 Localscan（并发）过程中被锁住的记录数，文档上说明默认为 64；

LongMessageBuffer：这个参数定义的是消息传递时候的 buffer 大小，而这里的消息传递主要是内部信息传递以及节点与节点之间的信息传递。这个参数一般很少需要调整，默认大小为 1MB 大小；

下面介绍一下与 log 相关的参数配置说明，包括 log level。这里的 log level 有多种，从 0 到 15，也就是共 16 种。如果设定为 0，则表示不记录任何 log。如果设置为最高 level，也就是 15，则表示所有的信息都会通过标准输出来记录 log。由于这里的所有信息实际上都会传递到管理节点的 cluster log 中，所以，一般来说，除了启动时候的 log 级别需要设置为 1 之外，其他所有的 log level 都只需要设置为 0 就可以了。

NoOfFragmentLogFiles：这个参数实际上和 Oracle 的 redo log 的 group 一样的。其实就是 ndb 的 redo log group 数目，这些 redo log 用于存放 ndb 引擎所做的所有需要变更数据的事情，以及各种 checkpoint 信息等。默认值为 8；

MaxNoOfSavedMessages：这个参数设定了可以保留的 trace 文件（在节点 crash 的时候参数）的最大个数，文档上面说此参数默认值为 25。

LogLevelStartup：设定启动 ndb 节点时候需要记录的信息的级别（不同级别所记录的信息的详细程度不一样），默认级别为 1；

LogLevelShutdown：设定关闭 ndb 节点时候记录日志的信息的级别，默认为 0；

LogLevelStatistic：这个参数是针对于统计相关的日志的，就像更新数量，插入数量，buffer 使用情况，主键数量等等统计信息。默认日志级别为 0；

LogLevelCheckpoint：checkpoint 日志记录级别（包括 local 和 global 的），默认为 0；

LogLevelNodeRestart：ndb 节点重启过程日志级别，默认为 0；

LogLevelConnection：各节点之间连接相关日志记录的级别，默认 0；

LogLevelError：在整个 Cluster 中错误或者警告信息的日志记录级别，默认 0；

LogLevelInfo：普通信息的日志记录级别，默认为 0。

这里再介绍几个用来作为 log 记录时候需要用到的 Buffer 相关参数，这些参数对于性能都有一定的影响。当然，如果节点运行在无盘模式下的话，则影响不大。

UndoIndexBuffer：undo index buffer 主要是用于存储主键 hash 索引在变更之后产生的 undo 信息的缓冲区。默认值为 2M 大小，最小可以设置为 1M，对于大多数应用来说，2M 的默认值是够的。当然，在更新非常频繁的应用里面，适当的调大此参数值对性能还是有一定帮助的。如果此参数太小，会报出 677 错误：Index UNDO buffers overloaded;

UndoDataBuffer：和 undo index buffer 类似，undo data buffer 主要是在数据发生变更的时候所需要的 undo 信息的缓冲区。默认大小为 16M，最小同样为 1M。当这个参数值太小的时候，系统会报出如下的错误：Data UNDO buffers overloaded，错误号为 891;

RedoBuffer：Redo buffer 是用 redo log 信息的缓冲区，默认大小为 8M，最小为 1M。如果此 buffer 太小，会报 1221 错误：REDO log buffers overloaded。

此外，NDB 节点还有一些和 metadata 以及内部控制相关的参数，但大部分参数都基本上不需要任何调整，所以就不做进一步介绍。如果有兴趣希望详细了解，可以根据 MySQL 官方的相关参考手册，手册上面都有较为详细的介绍。

3、SQL 节点相关配置说明

1）和其他节点一样，先介绍一些适用于所有节点的[MySQLD DEFAULT]参数

ArbitrationRank：这个参数在介绍管理节点的参数时候已经介绍过了，用于设定节点级别（主要是在多个节点在处理相关操作时候出现分歧时候设定裁定者）的。一般来说，所有的 SQL 节点都应该设定为 2;

ArbitrationDelay：默认为 0，裁定者在开始裁定之前需要被 delay 多久，单位为毫秒。一般不需要更改默认值。

BatchByteSize：在做全表扫描或者索引范围扫描的时候，每一次 fatch 的数据量，默认为 32KB;

BatchSize：类似 BatchByteSize 参数，只不过 BatchSize 所设定的是每一次 fetch 的 record 数量，而不是物理总量，默认为 64，最大为 992（暂时还不知道这个值是基于什么理论而设定的）。在实际运行 query 的过程中，fetch 的量受到 BatchByteSize 和 BatchSize 两个参数的共同制约，二者取最小值;

MaxScanBatchSize：在 Cluster 环境中，进行并行处理的情况下，所有节点的 BatchSize 总和的最大值。默认值为 256KB，最大值为 16MB。

2）每个节点独有的[MySQLD]参数组，仅有 id 和 hostname 参数需要配置，在之前各类节点均有介绍了，这里就不再累述。

16.4 MySQL Cluster 基本管理与维护

MySQL Cluster 的管理和普通的 MySQL Server 管理区别较大，基本上大部分管理工作都是在管理节点上面完成，仅有少数管理内容需要在其他节点实施。

1、各节点启动与关闭

要想 Cluster 环境能够正常工作，只好要启动一个 NDB 节点和一个 SQL 节点，另外为了完成管理，也至少要启动一个管理节点。各类节点的启动顺序也有要求，首先是管理节点，然后是 NDB 节点，最后才是 SQL 节点。

1) 按顺序启动各节点：

a、 启动管理节点：

[root@localhost MySQL-cluster]# ndb_mgmd -f /var/lib/MySQL-cluster/config.ini

这里执行的 ndb_mgmd 命令实际上就是 MySQL Cluster 管理服务器，可以通过-f config_file_name 或者--config=config_filename 来指定 MySQL Cluster 集群的参数文件。如果想了解更多关于 ndb_mgmd 的参数信息，可以通过运行 ndb_mgmd --help 来获取更详细的信息。

b、 启动用于存储数据的 ndb 节点

要启动存储节点，必须在每一台 ndb 节点主机上面都执行 ndbd 程序，如果是第一次启动，则需要添加--initial 参数，以便进行 ndb 节点的初始化工作。但是，在以后的启动过程中，是不能添加该参数的，否则 ndbd 程序会清除在之前建立的所有用于恢复的数据文件和日志文件。启动命令如下

root@ndb1:/root>ndbd --initial

c、 启动 SQL 节点

SQL 节点的启动和普通 MySQL Server 的启动没有太多明显的差别，不过有一个前提就是需要在 MySQL Server 的配置文件 my.cnf 设置好[MySQL_cluster]配置组中的 ndb-connectstring 参数和[MySQLd]配置组中的 ndbcluster 参数。

root@mysql1:/root>MySQLd_safe --user=MySQL &

2) 节点状态检查：

在各节点都启动完成后，回到管理节点，可以通过 ndb_mgm 来查看各节点状态：

```
[root@localhost MySQL-cluster]# ndb_mgm -e SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @192.168.0.3  (Version: 5.0.51, Nodegroup: 0, Master)
id=3    @192.168.0.4  (Version: 5.0.51, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.0.5  (Version: 5.0.51)
```

[MySQLd(API)]   2 node(s)
id=4   @192.168.0.1 (Version: 5.0.51)
id=5   @10.0.65.203 (Version: 5.0.51)

这里显示出整个集群有 5 个几点，其中各节点信息如下：

a) 2 个 NDBD 节点：

[ndbd(NDB)]   2 node(s)
id=2   @192.168.0.3 (Version: 5.0.51, Nodegroup: 0, Master)
id=3   @192.168.0.4 (Version: 5.0.51, Nodegroup: 0)

b) 两个 SQL 节点:

[MySQLd(API)]   2 node(s)
id=4   @192.168.0.1 (Version: 5.0.51)
id=5   @10.0.65.203 (Version: 5.0.51)

c) 1 个管理节点:

[ndb_mgmd(MGM)] 1 node(s)
id=1   @192.168.0.5 (Version: 5.0.51)

3) 节点的关闭操作：

在 MySQL Cluster 环境中，NDB 节点和管理节点的关闭都可以在管理节点的管理程序中完成，但是 SQL 节点却没办法。所以，在关闭整个 MySQL Cluster 环境或者关闭某个 SQL 节点的时候，首先必须到 SQL 节点主机上来关闭 SQL 节点程序。关闭方法和 MySQL Server 的关闭一样，就不累述。而 NDB 节点和管理节点则都可以在管理节点通过管理程序来完成：

```
ndb_mgm> shutdown
Connected to Management Server at: localhost:1186
Node 3: Cluster shutdown initiated
Node 2: Cluster shutdown initiated
Node 2: Node shutdown completed.
Node 3: Node shutdown completed.
2 NDB Cluster node(s) have shutdown.
Disconnecting to allow management server to shutdown.
```

2、基本管理维护

前面运行的命令 ndb_mgm 如果不带任何参数，实际上是进入 MySQL Cluster 的命令行管理界面。在命令行管理界面里面可以做大量的维护工作，如下：

```
[root@localhost MySQL-cluster]# ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm>
```

然后同样执行 show 命令：

```
ndb_mgm>show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
```

```
id=2 (not connected, accepting connect from 192.168.0.3)
id=3    @192.168.0.4  (Version: 5.0.51, Nodegroup: 0, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.0.5  (Version: 5.0.51)

[MySQLd(API)]   2 node(s)
id=4    @192.168.0.1  (Version: 5.0.51)
id=5    @10.0.65.203  (Version: 5.0.51)
```

我们可以看到结果和上面的完全一样。可以通过在 ndb 控制界面下执行 help 命令查看可以查看很多基本的维护管理命令：

```
ndb_mgm> help
---------------------------------------------------------------------------
 NDB Cluster -- Management Client -- Help
---------------------------------------------------------------------------
HELP                                   Print help text
HELP COMMAND                           Print detailed help for COMMAND(e.g.
SHOW)
SHOW                                   Print information about cluster
START BACKUP [NOWAIT | WAIT STARTED | WAIT COMPLETED]
                                       Start backup (default WAIT COMPLETED)

ABORT BACKUP <backup id>               Abort backup
SHUTDOWN                               Shutdown all processes in cluster
CLUSTERLOG ON [<severity>] ...         Enable Cluster logging
CLUSTERLOG OFF [<severity>] ...        Disable Cluster logging
CLUSTERLOG TOGGLE [<severity>] ...     Toggle severity filter on/off
CLUSTERLOG INFO                        Print cluster log information
<id> START                             Start data node (started with -n)
<id> RESTART [-n] [-i]                 Restart data or management server
node
<id> STOP                              Stop data or management server node
ENTER SINGLE USER MODE <id>            Enter single user mode
EXIT SINGLE USER MODE                  Exit single user mode
<id> STATUS                            Print status
<id> CLUSTERLOG {<category>=<level>}+  Set log level for cluster log
PURGE STALE SESSIONS                   Reset reserved nodeid's in the mgmt
server
CONNECT [<connectstring>]              Connect to management server
(reconnect if already connected)
QUIT                                   Quit management client
```

```
<severity> = ALERT | CRITICAL | ERROR | WARNING | INFO | DEBUG
<category> = STARTUP | SHUTDOWN | STATISTICS | CHECKPOINT | NODERESTART | CONNECTION | INFO | ERROR | CONGESTION | DEBUG | BACKUP
<level>    = 0 - 15
<id>       = ALL | Any database node id

For detailed help on COMMAND, use HELP COMMAND.
```

也可以通过执行help后面跟命令名称而获取各种命令的操作说明帮助信息：

```
ndb_mgm> help start
---------------------------------------------------------------------------
 NDB Cluster -- Management Client -- Help for START command
---------------------------------------------------------------------------
START  Start data node (started with -n)

<id> START          Start the data node identified by <id>.
                    Only starts data nodes that have not
                    yet joined the cluster. These are nodes
                    launched or restarted with the -n(--nostart)
                    option.

                    It does not launch the ndbd process on a remote
                    machine.
ndb_mgm> help shutdown
---------------------------------------------------------------------------
 NDB Cluster -- Management Client -- Help for SHUTDOWN command
---------------------------------------------------------------------------
SHUTDOWN  Shutdown the cluster

SHUTDOWN            Shutdown the data nodes and management nodes.
                    MySQL Servers and NDBAPI nodes are currently not
                    shut down by issuing this command.
ndb_mgm> help PURGE STALE SESSIONS
---------------------------------------------------------------------------
 NDB Cluster -- Management Client -- Help for PURGE STALE SESSIONS command
---------------------------------------------------------------------------
PURGE STALE SESSIONS  Reset reserved nodeid's in the mgmt server

PURGE STALE SESSIONS
                    Running this statement forces all reserved
                    node IDs to be checked; any that are not
                    being used by nodes acutally connected to
                    the cluster are then freed.
```

This command is not normally needed, but may be
required in some situations where failed nodes
cannot rejoin the cluster due to failing to
allocate a node id.

通过上面的几个帮助命令所获取的信息得知，我们可以通过在管理节点上面通过执行 restart，stop，shutdown 等基本的命令来重启某个节点，关闭某个节点，还可以同时一次性关闭所有节点。

此外，还可以通过执行备份相关的命令在管理节点对整个 Cluster 环境进行备份，以及通过日志相关命令实施对日志的相关管理。

16.5 基本优化思路

MySQL Cluster 虽然是一个分布式的数据库系统，但是在大部分地方的优化思路和方法还是和普通的 MySQL Server 一样。和常规 MySQL Server 在优化方面的区别主要提现在各节点之间的协作配置以及网络环境相关的优化。

由于 MySQL Cluster 是一个分布式的环境，而且所有访问都是需要经过超过一个节点（至少有一个 SQL 节点和一个 NDB 节点）才能完成，所以各个节点之间的协作配合就显得尤为重要。

首先，由于各个节点之间存在大量的数据通讯，所以节点之间的内部互联网络带宽一定要保证足够使用。为了适应不同的网络环境和性能需求，MySQL Cluster 支持了多种内部网络互联的协议和方式。最为常用的自然是通过 TCP/IP 来进行互联。此外还可以有 SCI Socket 方式来进行互联，还支持 Myrinet，Infiniband，VIA 接口等等。

其次，SQL 节点和 NDB 节点的主机性能配比应该合适，而不应该出现某一类节点过早出现瓶颈的时候，另外一类节点却还处于非常空闲的状态。如果在我们遇到的环境中出现这样的情况，那么我们就该重新评估两类节点的硬件设备配比了。否则，有一类节点的硬件资源就相当处于浪费状态了。

最后，就是 SQL 节点 和 NDB 节点两者软件配置方面的优化了。对于 SQL 节点的配置，和普通的 MySQL 区别不是太大。各类参数的配置原则也和普通 MySQL 基本相同。NDB Cluster 存储引擎的主要配置参数在前面的配置介绍中也基本都进行了性能相关的说明，这里就不再累述了。

16.6 小结

MySQL Cluster 的核心在于 NDB Cluster 存储引擎，他不仅对数据进行了水平切分，

还对数据进行了跨节点冗余。既解决了数据库的扩展问题，同时也在很大程度上提高了数据库整体可用性。

虽然目前 MySQL Cluster 的应用还不如普通的 MySQL Server 应用那么广泛，但是我想随着他的不断成熟和改善，将会被越来越广泛的使用。这种 Share Nothing 的 Cluster 架构也很可能会成为未来的趋势，就让我们共同期待越来越成熟稳定高效的 MySQL Cluster 吧。

第 17 章 高可用设计之思路及方案

前言：

数据库系统是一个应用系统的核心部分，要想系统整体可用性得到保证，数据库系统就不能出现任何问题。对于一个企业级的系统来说，数据库系统的可用性尤为重要。数据库系统一旦出现问题无法提供服务，所有系统都可能无法继续工作，而不像软件中部分系统出现问题可能影响的仅仅只是某个功能无法继续服务。所以，一个成功的数据库架构在高可用设计方面也是需要充分考虑的。本章内容将针对如何构建一个高可用的 MySQL 数据库系统来介绍各种解决方案以及方案之间的比较。

17.1 利用 Replication 来实现高可用架构

做维护的读者朋友应该都清楚，任何设备（或服务），只要是单点，就存在着很大的安全隐患。因为一旦这台设备（或服务） crash 之后，在短时间内就很难有备用设备（或服务）来顶替其功能。所以稍微重要一些的服务器或者应用系统，都会存在至少一个备份以供出现异常的时候能够很快的顶替上来提供服务。

对于数据库来说，主备配置是非常常见的设计思路。而对于 MySQL 来说，可以说天生

就具备了实现该功能的优势，因为其 Replication 功能在实际应用中被广泛的用来实现主备配置的功能。

我想，在大家所接触的 MySQL 环境中大多数都存在通过 MySQL Replication 来实现两台（或者多台）MySQL Server 之间的数据库复制功能吧。可能有些是为了增强系统扩展性，满足性能要求实现读写分离。亦或者就是用来作为主备机的设计，保证当主机 crash 之后在很短的时间内就可以将应用切换到备机上面来继续运行。

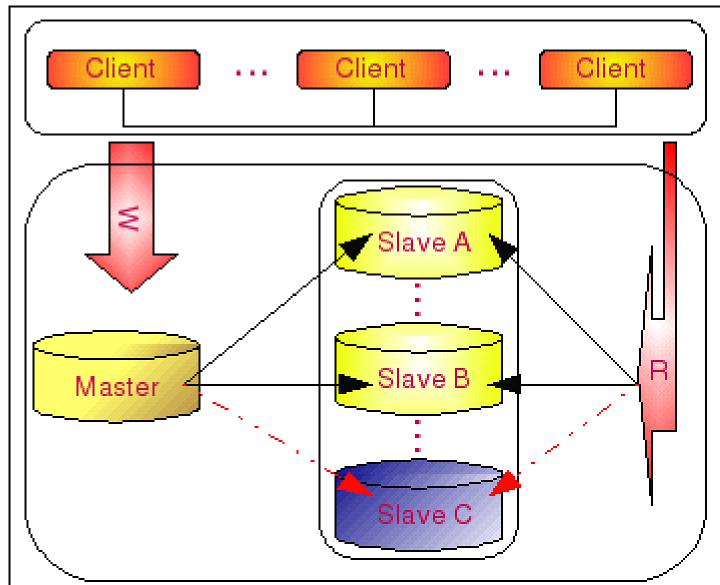
通过 MySQL Replication 来实现数据库的复制实际上在前面第 13 章中的内容中已经做过详细的介绍了，也介绍了多种 Replication 架构的实现原理及实现方法。在之前，主要是从扩展性方面来介绍 Replication。在这里，我将主要介绍的是从系统高可用方面如何和利用 Replication 的多种架构实现解决高可靠性的问题。

17.1.1 常规的 Master - Slave 解决基本的主备设计

在之前的章节中我提到过，普通的 Master - Slave 架构是目前很多系统中使用最为常见的一种架构方式。该架构设计不仅仅在很大程度上解决的系统的扩展性问题，带来性能的提升，同时在系统可靠性方面也提供了一定的保证。

在普通的一个 Master 后面复制一个或者多个 Slave 的架构设计中，当我们的某一台 Slave 出现故障不能提供服务之后，我们还有至少一台 MySQL 服务器（Master）可以提供服务，不至于所有和数据库相关的业务都不能运行下去。如果 Slave 超过一台，那么剩下的 Slave 也仍然能够不受任何干扰的继续提供服务。

当然，这里有一点在设计的时候是需要注意的，那就是我们的 MySQL 数据库集群整体的服务能力要至少能够保证当其缺少一台 MySQL Server 之后还能够支撑系统的负载，否则一切都是空谈。不过，从系统设计角度来说，系统处理能力留有一定的剩余空间是一个比较基本的要求。所以正常来说，系统中短时间内少一台 MySQL Server 应该是仍然能够支撑正常业务的。



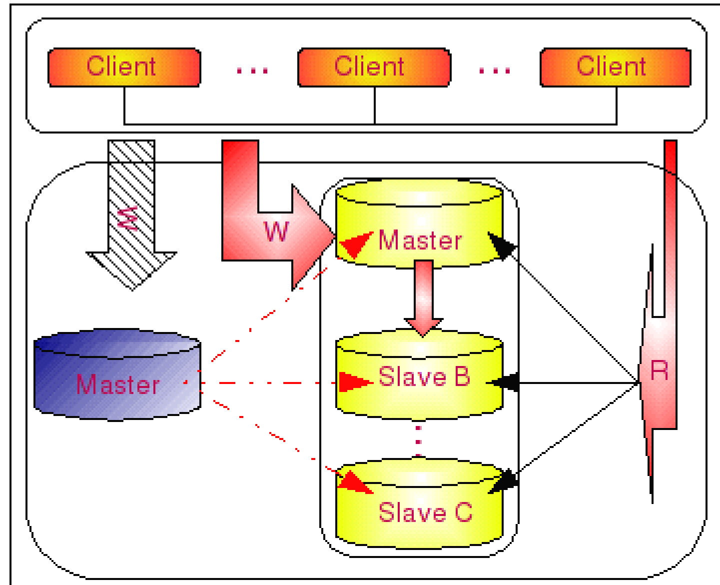
如上图所示，当我们的 Slave 集群中的一台 Slave C 出现故障 crash 之后，整个系统的变化仅仅只是从 Master 至 Slave C 的复制中断，客户端应用的 Read 请求也不能再访问 Slave C。当时其他所有的 MySQL Server 在不需要任何调整的情况下就能正常工作。客户端的请求 Read 请求全部由 Slave A 和 Slave B 来承担。

17.1.2 Master 单点问题的解决

上面的架构可以很容易的解决 Slave 出现故障的情况，而且不需要进行任何调整就能继续提供服务。但是，当我们的 Master 出现问题后呢？当我们的 Master 出现问题之后所有客户端的 Write 请求就都无法处理了。

这时候我们可以有如下两种解决方案，一个是将 Slave 中的某一台切换成 Master 对外提供服务，同时将其他所有的 Slave 都以通过 CHANGE MASTER 命令来将通过新的 Master 进行复制。另一个方案就是新增一台 Master，也就是 Dual Master 的解决方案。

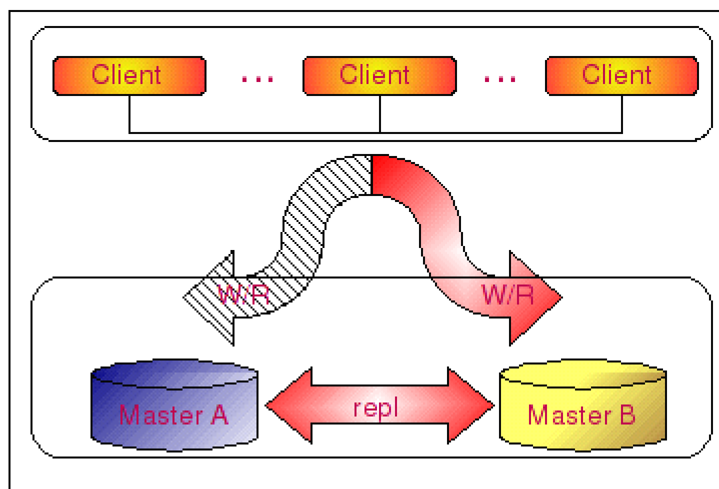
我们先来看看第一种解决方案，将一台 Slave 切换成 Master 来解决问题，如图：



当 Master 出现故障 crash 之后，原客户端对 Master 的所有 Write 请求都会无法再继续进行下去了，所有原 Master 到 Slave 的复制也自然就断掉了。这时候，我们选择一台 Slave 将其切换成 Master。假设选择的是 Slave A，则我们将其他 Slave B 和 Slave C 都通过 CHANGE MASTER TO 命令更换其 Master，从新的 Master 也就是原 Slave A 开始继续进行复制。同时将应用端所有的写入请求转向到新的 Master。对于 Read 请求，我们可以去掉对新 Master 的 Read 请求，也可以继续保留。

这种方案最大的一个弊端就是切换步骤比较多，实现比较复杂。而且，在 Master 出现故障 crash 的那个时刻，我们的所有 Slave 的复制进度并不一定完全一致，有可能有少量的差异。这时候，选择哪一个 Slave 作为 Master 也是一个比较头疼的问题。所以这个方案的可控性并不是特别的高。

我们再来看看第二种解决方案，也就是通过 Dual Master 来解决 Master 的点问，为了简单明了，这里就仅画出 Master 部分的图，如下：



我们通过两台 MySQL Server 搭建成 Dual Master 环境，正常情况下，所有客户端的 Write 请求都写往 Master A，然后通过 Replication 将 Master A 复制到 Master B。一旦 Master A 出现问题之后，所有的 Write 请求都转向 Master B。而在正常情况下，当 Master B 出现问题的时候，实际上不论是数据库还是客户端的请求，都不会受到实质性的影响。

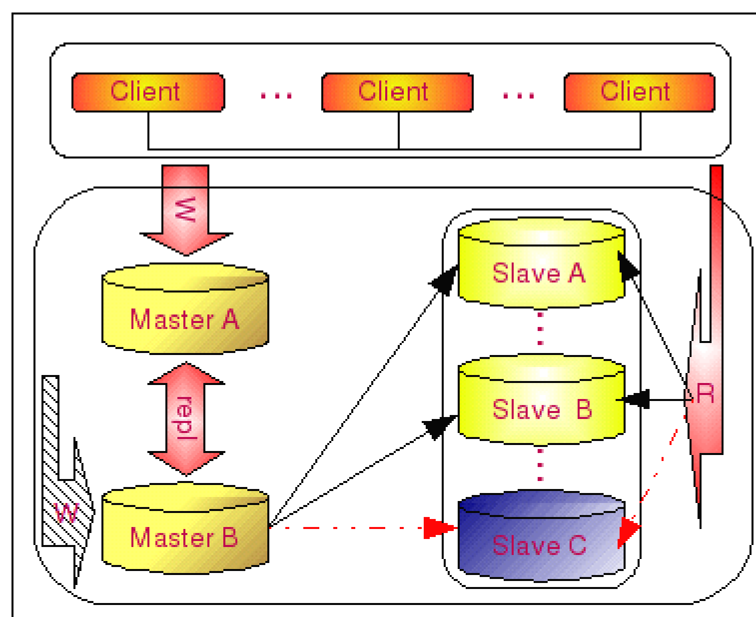
这里，可能有读者朋友会想到，当我们的 Master A 出现问题的时候，应用如何做到自动将请求转向到 Master B 呢？其实很简单，我们只需要通过相应的硬件设备如 F5 或者 Cluster 管理软件如 Heartbeat 来设置一个 VIP，正常情况下该 VIP 指向 Master A，而一旦 Master A 出现异常 crash 之后，则自动切换指向到 Master B，前端所的应用都通过这个 VIP 来访问 Master。这样，既解决了应用的 IP 切换问题，还保证了在任何时刻应用都只会见到一台 Master，避免了多点写入出现数据紊乱的可能。

这个方案最大的特点就是在 Master 出现故障之后的处理比较简单，可控性比较大。而弊端就是需要增加一台 MySQL 服务器，在成本方面投入更大。

17.1.3 Dual Master 与级联复制结合解决异常故障下的高可用

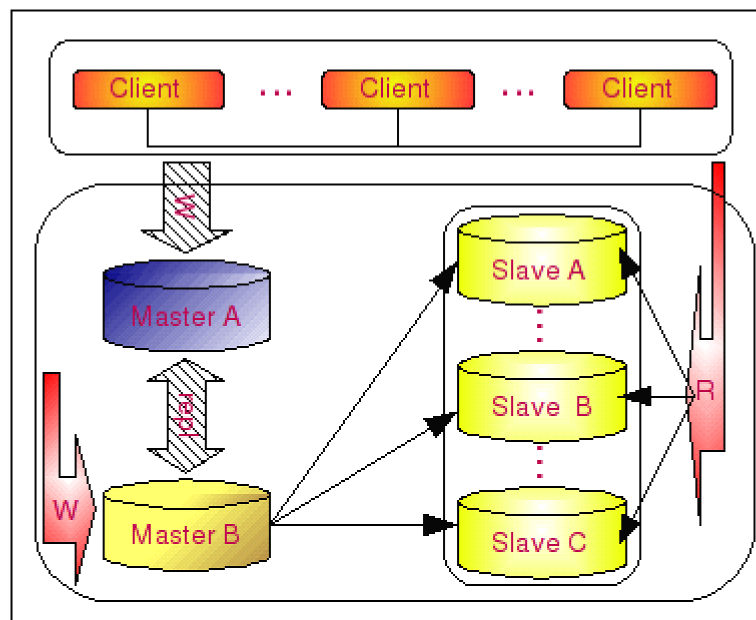
通过前面的架构分析，我们分别得到了 Slave 出现故障后的解决方案，也解决了 Master 的单点问题。现在我们再通过 Dual Master 与级联复制结合的架构，来得到一个整体的解决方案，解决系统整体可靠性的问题。

这个架构方案的介绍在之前的章节中已经详细的描述过了，这里我们主要分析一下处于高可靠性方面考虑的完善和异常切换方法。



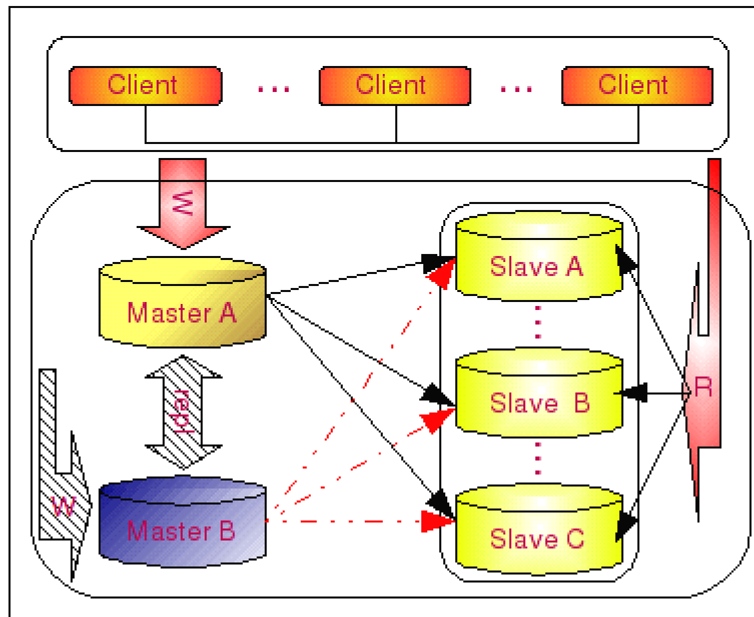
如上图所示，首先考虑 Slave 出现异常的情况。在这个架构中，Slave 出现异常后的处理情况和普通的 Master - Slave 架构的处理方式完全一样，仅仅需要在应用访问 Slave 集群的访问配置中去掉一个 Slave 节点即可解决，不论是通过应用程序自己判断，还是通过硬件解决方案如 F5 都可以很容易的实现。

下面我们再看看当 Master A 出现故障 crash 之后的处理方案。如下图：



当 Master A 出现故障 crash 之后，Master A 与 Master B 之间的复制将中断，所有客户端向 Master A 的 Write 请求都必须转向 Master B。这个转向动作的实现，可以通过上一节中所介绍的第二中方案中所介绍的通过 VIP 的方式实现。由于之前所有的 Slave 就都是从 Master B 来实现复制，所以 Slave 集群不会受到任何的影响，客户端的所有 Read 请求也就不会受到任何的影响，整个过程可以完全自动进行，不需要任何的人为干预。不过这里有一个隐患就是当 Master A crash 的时候如果 Master B 作为 Slave 的 IO 线程如果还没有读取完 Master A 的二进制日志的话，就会出现数据丢失的问题。要完全解决这个问题，我们只能通过第三方 patch（google 开发)来镜像 MySQL 的二进制日志到 Master B 上面，才能完全避免不丢失任何数据。

那么当 Master B 出现故障 crash 之后的情况又如何呢？如下图所示：



如果出现故障的不是 Master B 而是 Master A 又会怎样呢？首先可以确定的是我们的所有 Write 请求都不会受到任何影响，而且所有的 Read 请求也都还是能够正常访问。但所有 Slave 的复制都会中断，Slave 上面的数据会开始出现滞后的现象。这时候我们需要做的就是将所有的 Slave 进行 CHANGE MASTER TO 操作，改为从 Master A 进行复制。由于所有 Slave 的复制都不可能超前最初的数据源，所以可以根据 Slave 上面的 Relay Log 中的时间戳信息与 Master A 中的时间戳信息进行对照来找到准确的复制起始点，不会造成任何的数据丢失。

17.1.4 Dual Master 与级联复制结合解决在线 DDL 变更问题

当我们使用 Dual Master 加级联复制的组合架构的时候，对于 MySQL 的一个致命伤也就是在线 DDL 变更来说，也可以得到一定的解决。如当我们需要给某个表 tab 增加一个字段，可以通过如下在上述架构中来实现：

1、 在 Slave 集群中抽出一台暂时停止提供服务，然后对其进行变更，完成后再放回集群继续提供服务；
2、 重复第一步的操作完成所有 Slave 的变更；
3、 暂停 Master B 的复制，同时关闭当前 session 记录二进制日志的功能，对其进行变更，完成后再启动复制；
4、 通过 VIP 切换，将应用所有对 Master A 的请求切换至 Master B；
5、 关闭 Master A 当前 session 记录二进制日志的功能，然后进行变更；
6、 最后再将 VIP 从 Master B 切换回 Master A，至此，所有变更完成。

变更过程中有几点需要注意的：

1、 整个 Slave 集群需要能够承受在少一台 MySQL 的时候仍然能够支撑所有业务；
2、 Slave 集群中增加或者减少一台 MySQL 的操作简单，可通过在线调整应用配置来

实现；

3、 Dual Master 之间的 VIP 切换简单，且切换时间较短，因为这个切换过程会造成短时间段内 应用无法访问 Master 数据库。
4、 在变更 Master B 的时候，会出现短时间段内 Slave 集群数据的延时，所以如果单台主机的变更时间较长的话，需要在业务量较低的凌晨进行变更。如果有必要，甚至可能需要变更 Master B 之前将所有 Slave 切换为以 Master B 作为 Master。

当然，即使是这样，由于存在 Master A 与 Master B 之间的 VIP 切换，我们仍然会出现短时间段内应用无法进行写入操作的情况。所以说，这种方案也仅仅能够在一定程度上面解决 MySQL 在线 DDL 的问题。而且当集群数量较多，而且每个集群的节点也较多的情况下，整个操作过程将会非常的复杂也很漫长。对于 MySQL 在线 DDL 的问题，目前也确实还没有一个非常完美的解决方案，只能期待 MySQL 能够在后续版本中尽快解决这个问题了。

17.2 利用 MySQL Cluster 实现整体高可用

在上一章中已经详细介绍过 MySQL Cluster 的相关特性，以及安装配置维护方面的内容。这里，主要是介绍一下如何利用 MySQL Cluster 的特性来提高我们系统的整体可用性。

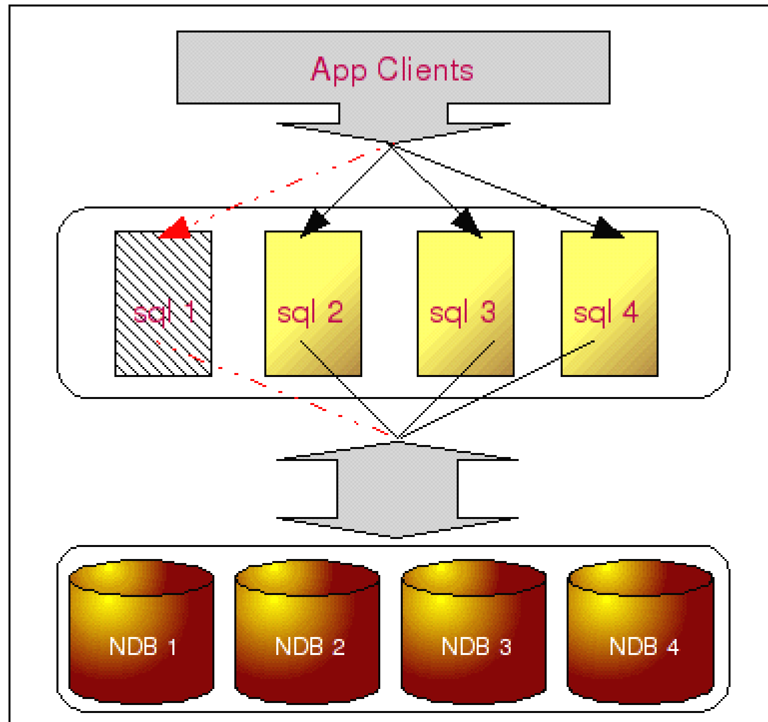
由于 MySQL Cluster 本身就是一个完整的分布式架构的系统，而且支持数据的多点冗余存放，数据实时同步等特性。所以可以说他天生就已经具备了实现高可靠性的条件了，是否能够在实际应用中满足要求，主要就是在系统搭建配置方面的合理设置了。

由于 MySQL Cluster 的架构主要由两个层次两组集群来组成,包括 SQL 节点(mysqld)和 NDB 节点（数据节点），所有两个层次都需要能够保证高可靠性才能保证整体的可靠性。下面我们从两个方面分别来介绍 MySQL Cluster 的高可靠性。

17.2.1 SQL 节点的高可靠性保证

MySQL Cluster 中的 SQL 节点实际上就是一个多节点的 mysqld 服务，并不包含任何数据。所以，SQL 节点集群就像其他任何普通的应用服务器一样，可替代性很高，只要安装了支持 MySQL Cluster 的 MySQL Server 端即可。

当该集群中的一个 SQL 节点 crash 掉之后，由于只是单纯的应用服务，所以并不会造成任何的数据丢失。只需要前端的应用数据源配置兼容了集群中某台主机 crash 之后自动将该主机从集群中去除就可以了。实际上，这一点对于应用服务器来说是非常容易做到的，无论是通过自行开发判断功能的代理还是通过硬件级别的负载均衡设备，都可以非常容易做到。当然，前提自然也是剩下的 SQL 节点能够承担整体负载才行。



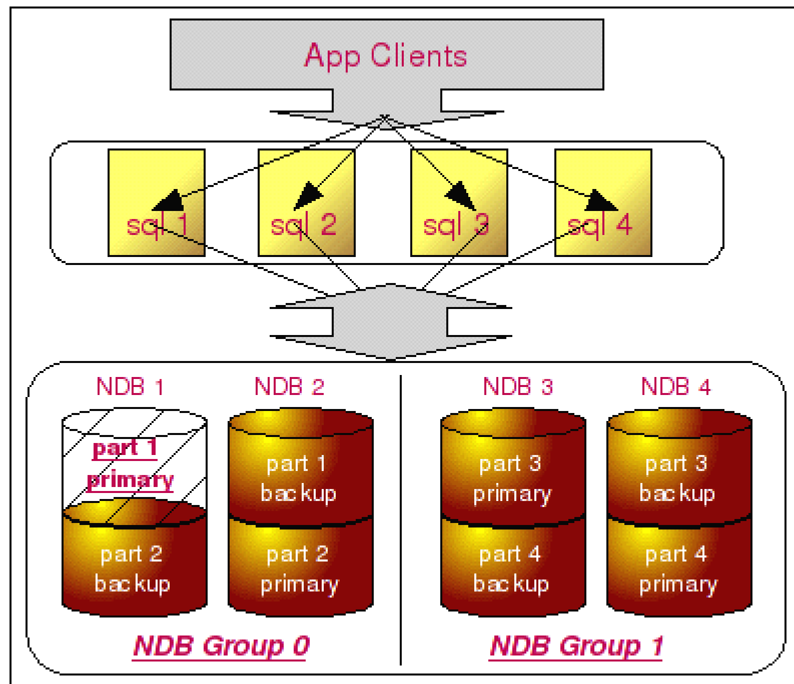
如上图，当 SQL 1 crash 之后，实际上仅仅只是访问到数据的很多条途径中的某一条中断了，实际上仍然还有很多条途径可以获取到所需要的数据。而且，由于 SQL 的可替代性很高，所以更换也非常简单，即使更换整台主机，也可以在短时间内完成。

17.2.2 NDB 节点的高可靠性保证

MySQL Cluster 的数据冗余是有一个前提条件的，首先必须要保证有足够的节点，实际上是至少需要 2 个节点才能保证数据有冗余，因为，MySQL Cluster 在保存冗余数据的时候，是比需要确保同一份数据的冗余存储在不同的节点之上。在保证了冗余的前提下，MySQL Cluster 才会将数据进行分区。

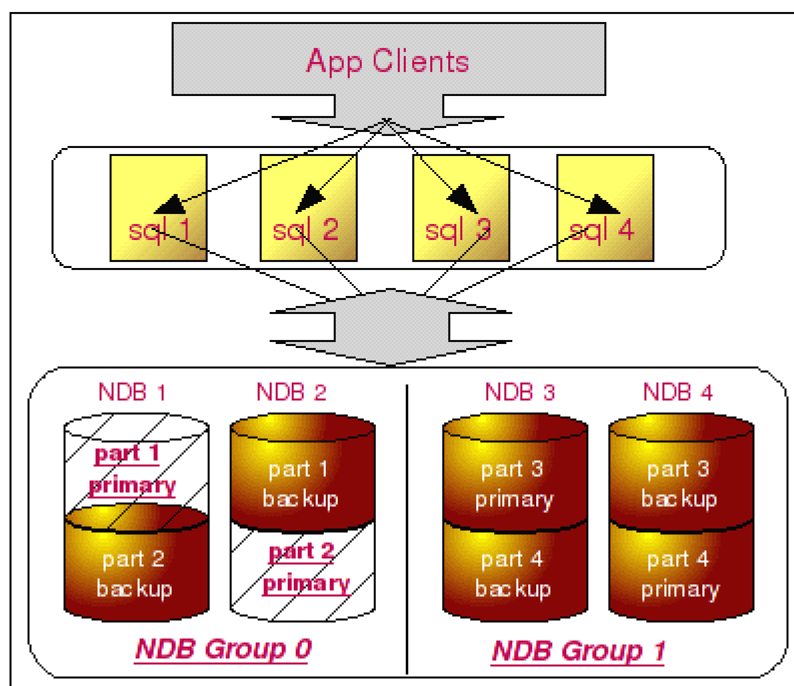
假设我们存在 4 个 NDB 节点，数据被分成 4 个 partition 存放，数据冗余存储，每份数据存储 2 份，也就是说 NDB 配置中的 NoOfReplicas 参数设置为 2，4 个节点将被分成 2 个 NDB Group。

所有数据的分布类似于下图所示：



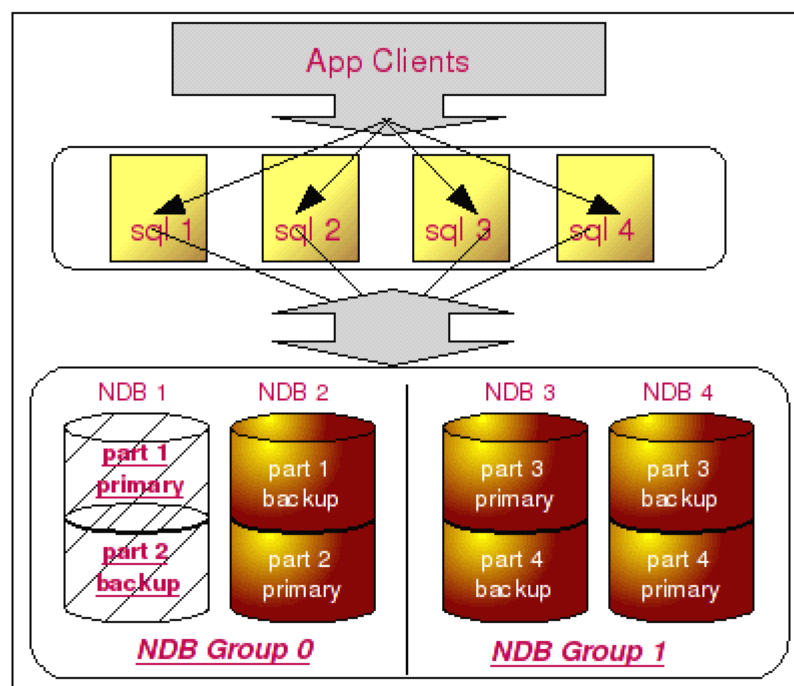
在这样的配置中，假设我们 NDB Group 0 这一组中的某一个 NDB 节点(假如是 NDB 1)出现问题，其中的部分数据（假设是 part 1）坏了，由于每一份数据都存在一个冗余拷贝，所以并不会对系统造成任何的影响，甚至完全不需要人为的操作，MySQL 就可以继续正常的提供服务。

假如我们两个节点上面都出现部分数据损坏的情况，结果会怎样？如下图：



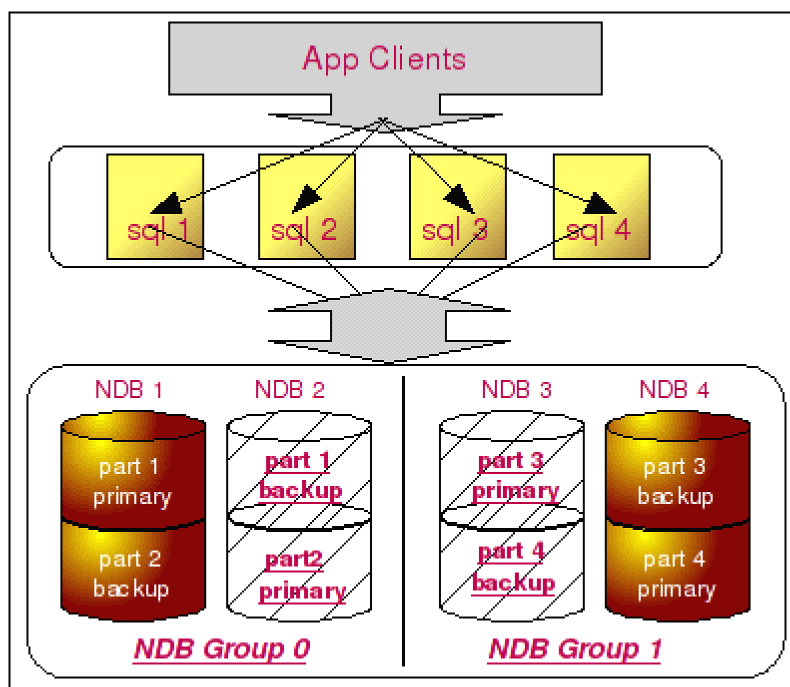
如果像上图所示，如果两个损坏部分数据的节点属于同一个 NDB Group，只要损坏部分并没有包含完全相同的数据，整个 MySQL Cluster 仍然可以正常提供服务。但是，如果损坏数据中存在相同的数据，即使只有很少的部分，都会造成 MySQL Cluster 出现问题，不能完全正常的提供服务。此外，如果损坏数据的节点处于两个不同的 NDB Group，那么非常幸运，不管损坏的是哪一部分数据，都不会影响 MySQL Cluster 的正常服务。

可能有读者朋友会说，那假如我们的硬件出现故障，整个 NDB 都 crash 了呢？是的，确实很可能存在这样的问题，不过我们同样不用担心，如图所示：



假设我们整个 NDB 节点由于硬件(或者软件)故障而 crash 之后，由于 MySQL Cluster 保证了每份数据的拷贝都不在同一台主机上，所以即使整太主机都 crash 了之后，MySQL Cluster 仍然能够正常提供服务，就像上图所示的那样，即使整个 NDB 1 节点都 crash 了，每一份数据都还可以通过 NDB 2 节点找回。

那如果是同时 crash 两个节点会是什么结果？首先可以肯定的是假如我们 crash 的两个节点处于同一个 NDB Group 中的话，那 MySQL Cluster 也没有办法了，因为两份冗余的数据都丢失了。但是只要 crash 的两个节点不在同一个 NDB Group 中，MySQL Cluster 就不会受到任何影响，还是能够继续提供正常服务。如下图所示的情况：



从上面所列举的情况我们可以知道，MySQL Cluster 确实可以达到非常高的可靠性，毕竟同一时刻存放相同数据的两个 NDB 节点都出现大故障的概率实在是太小了，要是这也能够被遇上，那只能自然倒霉了。

当然，由于 MySQL Cluster 之前的老版本需要将所有的数据全部 Load 到内存中才能正常运行，所有由于受到内存空间大小的限制，使用的人非常少。现在的新版本虽然已经支持仅仅只需要所有的索引数据 Load 到内存中即可，但是由于实际的成功案例还并不是很多，而且发展时间也还不是太长，所以很多用户朋友对于 MySQL Cluster 目前还是持谨慎态度，大部分还处于测试阶段。

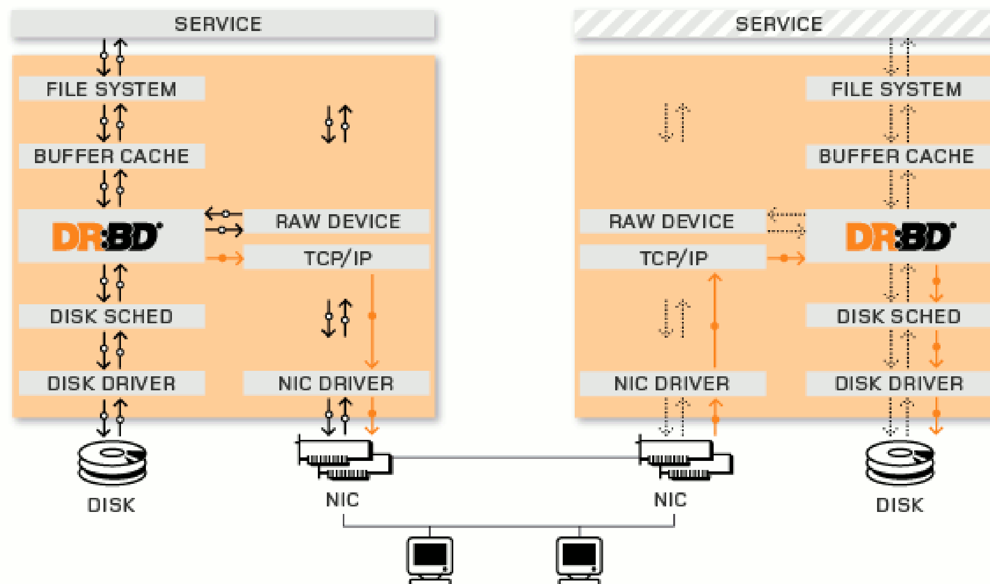
17.3 利用 DRBD 保证数据的高安全可靠

17.3.1 DRBD 介绍

对于很多多这朋友来说，DRBD 的使用可能还不是太熟悉，但多多少少可能有一些了解。毕竟，在 MySQL 的官方文档手册的 High Availability and Scalability 这一章中将 DRBD 作为 MySQL 实现高可用性的一个非常重要的方式来介绍的。虽然这一内容是直到去年年中的时候才开始加入到 MySQL 的文档手册中，但是 DRBD 本身在这之前很久就已经成为很多应用场合实现高可靠性的解决方案，而且在不少的 MySQL 使用群体中也早就开始使用了。

简单来说，DRBD 其实就是通过网络来实现块设备的数据镜像同步的一款开源 Cluster

软件，也被俗称为网络 RAID1。官方英文介绍为：DRBD refers to block devices designed as a building block to form high availability (HA) clusters. This is done by mirroring a whole block device via an assigned network. It is shown as network raid-1- DRBD。下面是 DRBD 的一个概览图：



从图中我们可以看出，DRBD 介于文件系统与磁盘介质之间，通过捕获上层文件系统的所有 IO 操作，然后调用内核中的 IO 模块来读写底层的磁盘介质。当 DRBD 捕获到文件系统的写操作之后，会在进行本地的磁盘写操作的同时，以 TCP/IP 协议将，通过本地主机的网络设备(NIC)将 IO 传递至远程主机的网络设备。当远程主机的 DRBD 监听到传递过来的 IO 信息之后，会立即将该数据写入到该 DRBD 所维护的磁盘设备。至此，整个 IO 才做完成。

实际上 DRBD 在处理远程数据写入的时候有三种复制模式（或者称为级别）可以选择，不同的复制模式保证了远程数据写入的三种可靠性。三种级别的选择可以通过 DRBD 的通用配置部分的 protocal。不同的复制模式，实际上是影响了一个 IO 完成所代表的实际含义。因为当我们使用 DRBD 的时候，一个 IO 完成的标识（DRBD 返回 IO 完成）是本地写入和远程写入这两个并发进程都返回完成标识。下面我来详细介绍一下这三种复制模式所代表的含义：

Protocol A：这种模式是可靠性最低的模式，而且是一个异步的模式。当我们使用这个模式来配置的时候，写远程数据的进程将数据通过 TCP/IP 协议发送进入本地主机的 TCP send buffer 中，即返回完成。

Protocol B：这种模式相对于 Protocol A 来说，可靠性要更高一些。因为写入远程的线程会等待网络信息传输完成，也就是数据已经被远程的 DRBD 接受到之后返回完成。

Protocol C：Protocol C 复制模式是真正完全的同步复制模式，只有当远程的 DRBD 将数据完全写入磁盘成功后，才会返回完成。

对比上面三种复制模式，C 模式可以保证不论出现任何异常，都能够保证两端数据的一致性。而如果使用 B 模式，则可能当远程主机突然断电之后，将丢失部分还没有完全写入磁盘的信息，且本地与远程的数据出现一定的不一致情况。当我们使用 A 复制模式的话，

可能存在的风险就要更大了。只要本地网络设备突然无法正常工作（包括主机断电），就会丢失将写入远程主机的数据，造成数据不一致现象。

由于不同模式所要求的远程写入进程返回完成信息的时机不一样，所以也直接决定了 IO 写入的性能。通过三个模式的描述，我们可以很清楚的知道，IO 写入性能与可靠程度成反比。所以，各位读者朋友在考虑设置这个参数的时候，需要仔细评估各方面的影响，尽可能得到一个既满足实际场景的性能需求，又能满足对可靠性的要求。

DRBD 的安装部署以及相关的配置说明，在 MySQL 的官方文档手册以及我的个人网站博客（http://www.jianzhaoyang.com）中都有相关的描述，而且在 DRBD 的官方网站上面也可以找到最为全面的说明，所以这里就不再介绍了。下面我主要介绍一下 DRBD 的一些特殊的特性以及限制，以供大家参考。

17.3.2 DRBD 特性与限制

DRBD 之所以能够得到广泛的采用，甚至被 MySQL 官方写入文档手册作为官方推荐的高可用方案之一，主要是其各种高可靠特性以及稳定性的缘故。下面介绍一下 DRBD 所具备的一些比较重要的特性。

1、 非常丰富的配置选项，可以适应我们的应用场景中的情况。无论是可靠性级别与性能的平衡，还是数据安全性方面，无论是本地磁盘，还是网络存储设备，无论是希望异常自动解决，还是希望手动控制等等，都可以通过简单的配置文件就可以解决。当然，丰富的配置在带来极大的灵活性的同时，也要求使用者需要对他有足够的了解才行，否则在那么多配置参数中也很难决策该如何配置。幸好 DRBD 在默认情况下的各项参数实际上就已经满足了大多数典型需求了，并不需要我们每一项参数都设置才能运行；

2、 对于节点之间出现数据不一致现象，DRBD 可以通过一定的规则，进行重新同步。而且可以通过相关参数配置让 DRBD 在固定时间点进行数据的校验比对，来确定数据是否一致，并对不一致的数据进行标记。同时还可以选择是 DRBD 自行解决不一致问题还是由我们手工决定如何同步；

3、 在运行过程中如果出现异常导致一端 crash，并不会影响另一端的正常工作。出现异常之后的所有数据变更，都会被记录到相关的日志文件中。当 crash 节点正常恢复之后，可以自动同步这段时间变更过的数据。为了不影响新数据的正常同步，还可以设定恢复过程中的速度，以确保网络和其他设备不会出现性能问题；

4、 多种文件系统类型的支持。DRBD 除了能够支持各种常规的文件系统之外，还可以支持 GFS，OCFS2 等分布式文件系统。而且，在使用分布式文件系统的时候，还可以实现各结带你同时提供所有 IO 操作。

5、 提供对 LVM 的支持。DRBD 既可以使用由 LVM 提供的逻辑设备，也可以将自己对外提供的设备成为 LVM 的物理设备，这极大的方便的运维人员对存储设备的管理。

6、 所有 IO 操作，能够绝对的保证 IO 顺序。这也是对于数据库来说非常重要的一个特性尤其是一些对数据一致性要求非常苛刻的数据库软件来说。

7、 可以支持多种传输协议，从 DRBD 8.2.7 开始，DRBD 开始支持 Ipv4，IPv6 以及 SuperSockets 来进行数据传输。

8、支持 Three-Way 复制。从 DRBD 8.3.0 开始，DRBD 可以支持三个节点之间的复制了，有点类似于级联复制的特性。

当然，DRBD 在拥有大量让人青睐的特性的同时，也存在一定的限制，下面就是 DRBD 目前存在的一些比较重要的限制：

1、对于使用常规文件系统（非分布式文件系统）的情况下，DRBD 只能支持单 Primary 模式。在单 Primary 模式下，只有一个节点的数据是可以对外提供 IO 服务的。只有当使用 GFS 或者 OCFS2 这样的分布式文件系统的时候，才能支持 Dual Primary 模式。
2、Sblit Brain 的解决。因为某些特殊的原因，造成两台主机之间的 DRBD 连接中断之后双方都以 Primary 角色来运行之后的处理还不是太稳定。虽然 DRBD 的配置文件中可以配置自动解决 Split Brain，但是从我之前的测试情况来看，并不是每次的解决都非常令人满意，在有些情况下，可能出现某个节点完全失效的可能。
3、复制节点数目的限制，即使是目前最新的 DRBD 版本来说，也最多只支持三个节点之间的复制。

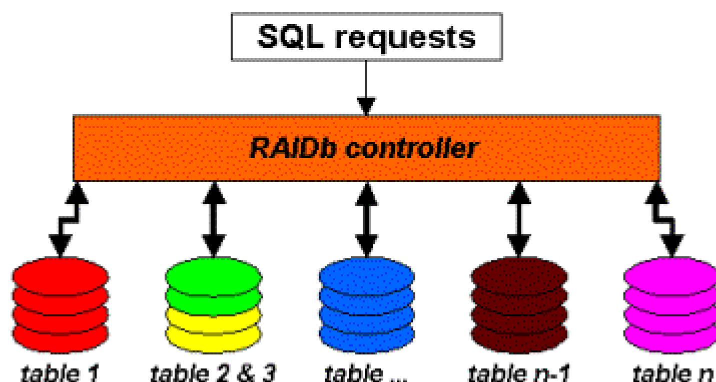
以上几个限制是在目前看来可能对使用者造成较大影响的几个限制。当然，DRBD 还存在很多其他方面的限制，大家在决定使用之前，还是需要经过足够测试了解，以确保不会造成上线后的困扰。

17.4 其他高可用设计方案

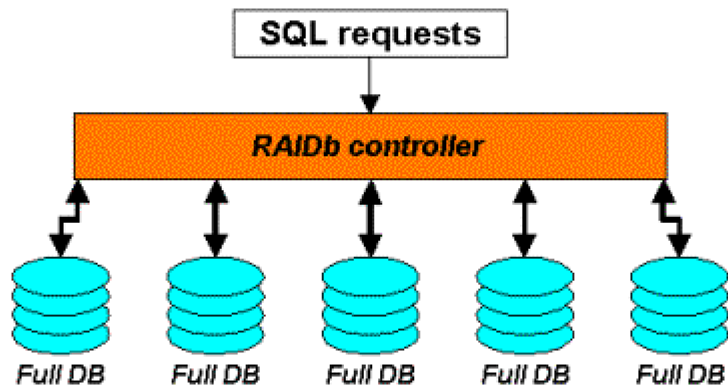
除了上面几种高可用方案之外，其实还有不少方案可以供大家选择，如 RaiDB，共享存储解（SAN 或者 NAS） 等等。

对于 RaiDB，可能很多读者朋友还比较陌生，其全称为 Redundant Arrays of Inexpensive Databases。也就是通过 Raid 理念来管理数据库的数据。所以 RaiDB 也存在数据库 Sharding 的概念。和磁盘 Raid 一样，RaiDB 也存在多种 Raid，如 RaiDB-0，RaiDB-1，RaiDB-2，RaiDB-0-1 和 RaiDB-1-0 等。商业 MySQL 数据库解决方案提供商 Continuent 的数据库解决方案中，就利用了 RaiDB 的概念。大家可以通过一下几张图片清晰的了解 RaiDB 的各种 Raid 模式。

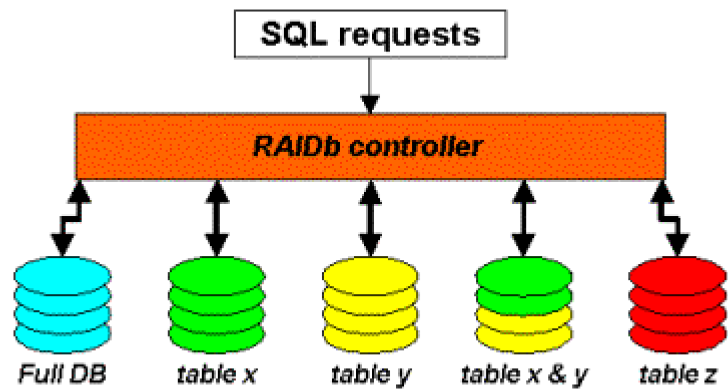
RaiDB-0:



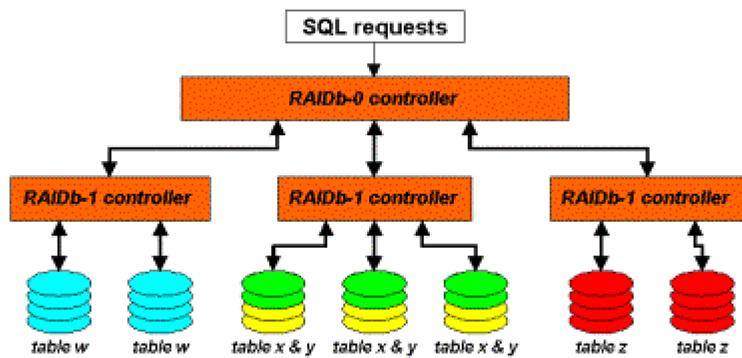
RaiDB-1:



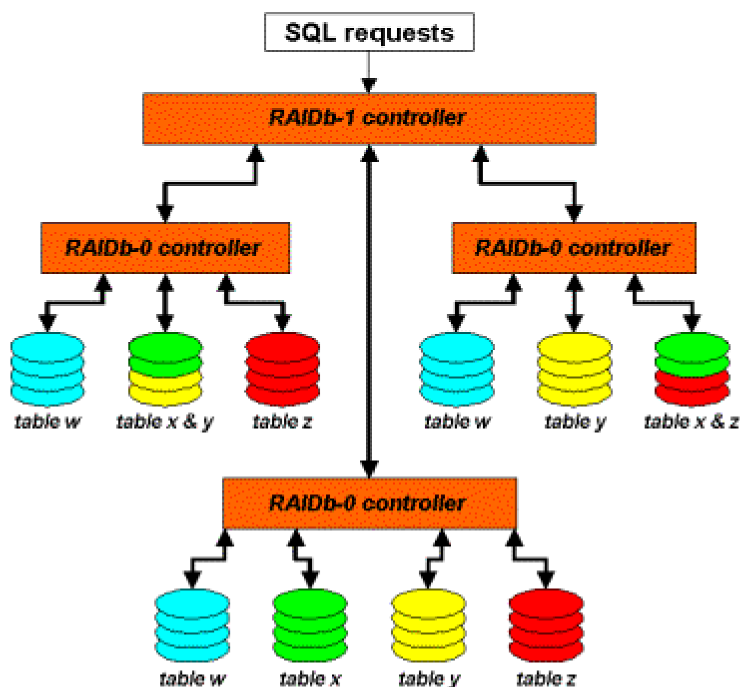
RaiDB-2:



RaiDB-0-1:



RaiDB-1-0:



从图中的标识，大家应该就比较清楚 RaiDB 各种 Raid 模式下的数据如何分布以及工作方式了吧。至于为什么这样作可以保证高可用性，就和磁盘通过 Raid 来保证数据高可靠性一样。当然，要真正理解，前提还是大家已经具备了清晰的 Raid 概念。

而对于共享存储的解决方案，则是一个相对来说比较昂贵的解决方案了。运行 MySQL Server 的主机上面并不存放数据，只不过通过共享协议（FC 或者 NFS）来将远程存储设备上面的磁盘空间 Mount 到本地，当作本地磁盘来使用。

为什么共享存储的解决方案可以满足高可靠性要求呢？首先，数据不是存在 MySQL Server 本地主机上面，所以当本地 MySQL Server 主机出现任何故障，都不会造成数据的丢失，完全可以通过其他主机来找回存储设备上面的数据。其次，无论是通过通过 SAN 还是 NAS 来作为共享存储，存储本身具备多种高可用解决方案，可以做到完全不丢失任何数据。甚至即使 MySQL Server 与共享存储之间的连接通道，也有很多成熟的高可用解决方案，来保证连接的高可用性。由于共享存储的解决方案本身违背了我个人提倡的通过 MySQL 构建廉价的企业级高性能高可靠性方案，又考虑到篇幅问题，所以这里就不详细深入介绍了。如果读者朋友对这方面比较感兴趣，可以通过其他图书再深入了解 SAN 存储以及 NAS 存储相关的知识。

17.5 各种高可用方案的利弊比较

从前面各种高可用设计方案的介绍中读者们可能已经发现，不管是哪一种方案，都存在自己独特的优势，但也都或多或少的存在一些限制。其实这也是很正常的，毕竟任何事物都不可能是完美的，我们只能充分利用各自的优势来解决自己的问题，而不是希望依赖某种方案一劳永逸的解决所有问题。这一节将针对上面的几种主要方案做一个利弊分析，以供大家

选择过程中参考。

1、 MySQL Replication

优势：部署简单，实施方便，维护也不复杂，是 MySQL 天生就支持的功能。且主备机之间切换方便，可以通过第三方软件或者自行编写简单的脚本即可自动完成主备切换。

劣势：如果 Master 主机硬件故障且无法恢复，则可能造成部分未传送到 Slave 端的数据丢失；

2、MySQL Cluster

优势：可用性非常高，性能非常好。每一分数据至少在不同主机上面存在一份拷贝，且冗余数据拷贝实时同步。

劣势：维护较为复杂，产品还比较新，存在部分 bug，目前还不一定适用于比较核心的线上系统。

3、DRBD 磁盘网络镜像方案

优势：软件功能强大，数据在底层快设备级别跨物理主机镜像，且可根据性能和可靠性要求配置不同级别的同步。IO 操作保持顺序，可满足数据库对数据一致性的苛刻要求。

劣势：非分布式文件系统环境无法支持镜像数据同时可见，性能和可靠性两者相互矛盾，无法适用于性能和可靠性要求都比较苛刻的环境。维护成本高于 MySQL Replication。

17.6 小结

本章重点针对 MySQL 自身具备的两种高可用解决方案以及 MySQL 官方推荐的 DRBD 解决方案做了较为详细的介绍，同时包括了各解决方案的利弊对比。希望这些信息能够给各位读者带来一些帮助。不过，MySQL 的高可用解决方案远远不只上面介绍过的集中方案，还存在着大量的其他方案可供各位读者朋友进行研究探索。开源的力量是巨大的，开源贡献者的力量更是无穷的。

第 18 章 高可用设计之 MySQL 监控

前言：

一个经过高可用可扩展设计的 MySQL 数据库集群，如果没有一个足够精细足够强大的监控系统，同样可能会让之前在高可用设计方面所做的努力功亏一篑。一个系统，无论如何设计如何维护，都无法完全避免出现异常的可能，监控系统就是根据系统的各项状态的分析，让我们能够尽可能多的提前预知系统可能会出现的异常状况。即使没有及时发现将要发生的异常，也要在异常出现后的第一时间知道系统已经出现异常，否则之前的设计工作很可能就白费了。

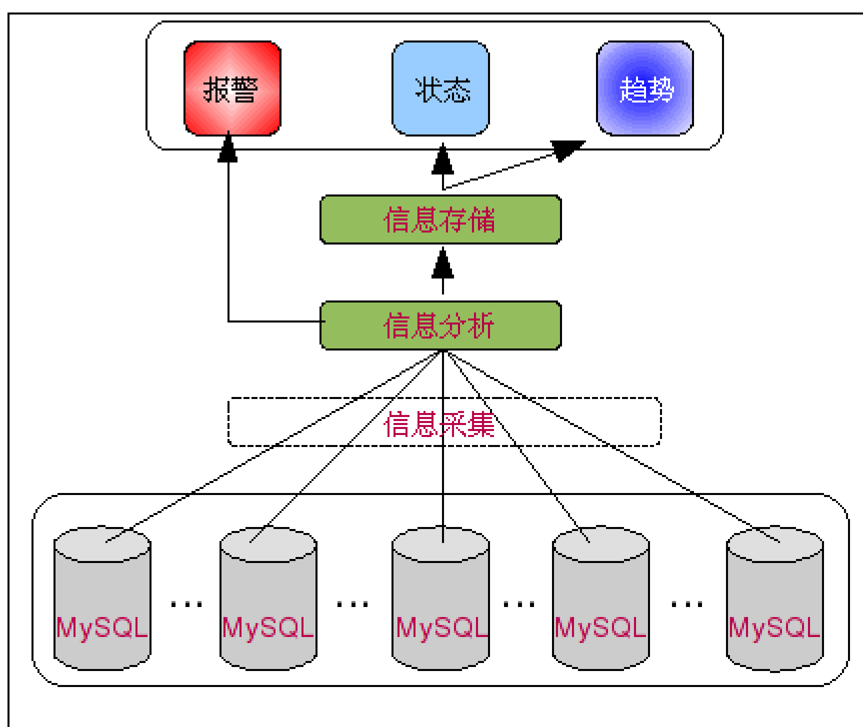
18.1 监控系统设计

系统监控在很多人眼中是一个没有多少技术含量的事情，其实并不是这样的。且不说一个大型网站的所有设备的监控，就仅仅只是搭建一个比较完善的几十台 MySQL 集群系统的监控，很可能就会让很多人束手无策，或者功能不够完善。

其实一个完善的大型集群系统的监控系统，和少数几台主机的监控是有很大差别的。少数机台主机的监控在大多数时候可以通过几个简单的脚本（Shell 或者 Perl 等），发发邮件，再高一点的报警信息发发手机短信，基本就搞定了。监控点少，发出的信息量也少。很多状态信息甚至都可以通过维护人员登录到主机上面来定时 Check 即可。可如果有上百台主机，仍然仅仅依靠在每台主机上面部署几个简单的脚本的方式来进行监控，后期的管理维护成本就会非常高了。

当 MySQL 主机达到一定规模之后，我们基本上很难通过人工到各个主机上面来定时检查各自的状态，不论是运行状态还是性能状态。甚至都不能像只有少数 MySQL 主机的时候那样简单的通过发送邮件的方式将相关信息发出。毕竟，量大了之后，检查邮件的时间成本也是很大的。这时候就需要我们进行统一的监控信息采集、分析、存储、处理系统来帮助我们过滤掉可以忽略的正常信息，并画出相关信息的趋势图，以帮助判断系统的运行状况和发展趋势。

所以，MySQL 分布式集群的监控系统整体架构体系应该如下图所示：



1、 信息采集

信息采集可以是一个统一的模块以主动的方式从各个 MySQL 主机上面来获取信息然后存放到监控信息存储中心，也可以是分布在各个 MySQL 主机上面的模块以被动的方式来反向将相关数据推向监控信息存储中心。

一般来说，较小规模的监控点可以采用轮询的方式主动采集数据，但是当监控点达到一定规模以后，轮询的主动采集方式可能就会遇到一定的性能瓶颈和信息延时问题，尤其是当需要采集的数据比较多的时候尤为突出。而如果要采用从各个 MySQL 节点进行被动的推送，则可能需要开发能够支持网络通信的监控程序，使采集的信息能够顺利的到达信息分析模块以即时得到分析，成本会稍微高一些。

不论是采用主动还是被动的方式来进行数据采集，我们都需要在监控主机上面部署采集相关信息的程序或脚本，包括主机信息和数据库信息。

2、 信息分析

当前端采集模块的监控程序获取到 MySQL 主机当前的各种状态信息之后，分析模块需要实时地对数据进行分析，识别出系统是正常还是异常，如果是异常，就必须通过相关机制立即发出报警通知，并将信息发送到存储模块进行持久化。如果正常，则不需要进行任何处理就可以将数据传递到存储模块持久化。对信息分析模块来讲，最重要的事情（？要求）就是处理及时、准确，当监控点到达一定的量之后，性能可能会成为瓶颈。

3、 信息存储

存储监控程序采集的信息也是一件非常重要的事情，因为不论是查看当前状态，还是绘制趋势图，都需要用到这些信息。此外还有一个非常重要的原因就是通过对积累下来的这些监控信息的分析挖掘，为系统的容量规划和性能模型设计带来非常大的帮助。

4、 信息处理

最后根据采集到的各种状态、性能信息，通过应用相应规则进行分析挖掘运算，绘制出各种状态的趋势图以供维护人员查看。此外，通过对各种趋势的分析，发掘出业务发展与数据库负载之间的关系，以及与主机硬件的关系。这些关系数据，对于系统发展规划的制定将是非常有意义的。

18.2 健康状态监控

健康状态信息一般来说还是比较简单的，但也是非常重要的。因为对于监控来说，需要了解的健康状态基本上也就只有“正常”或“不正常”这两种。下面分别从主机状态信息和数据库状态信息来分别介绍一下。

18.2.1 主机健康状态监控（？）

在数据库运行环境，需要关注的主机状态主要有网络通信、系统软硬件错误、磁盘空间、内存使用，进程数量等。

- 网络通信:网络通信基本上可以说是最容易检测的了,基本上只需要通过网络 ping 就可以获知是否正常。如果还不放心，或者所属网段内禁用了 ping，也可以从监控主机进行固定端口的 telnet 尝试或者 ssh 登录尝试。由于网络出现故障的时候被动的信息采集方式也会失效（无法与外界通信），所以网络通信的检测主要还是依靠主动检测。

- 系统软硬件错误:系统软硬件错误,一般只能通过检测各种日志文件的信息来实现，如主机的系统日志中基本上都会记录下 OS 能够检测到的大部分错误信息,如硬件错误，IO 错误等等。我们一般使用文本监控软件，如 sec、logwatch 等日志监控专用软件，通过配置相应的匹配规则，从日志文件中捕获满足条件的错误信息，再发送给信息分析模块。

- 磁盘空间：对于磁盘空间的使用状况监控，我们通过最简单的 shell 脚本就可以轻松搞定，得出系统中各个分区的当前使用量，剩余可用空间。积累一定时间段的信息之后，很容易就能得出系统数据量增长趋势。

- 内存使用：系统物理内存使用量的信息采集同样非常简单，只需要一个基本的系统命令“free”，就可以获得当前系统内存总量，剩余使用量，以及文件系统的 buffer 和 cache 两者使用量。而且，除了物理内存使用情况，还可以得到 swap 使用量。

通过 shell 脚本对这些输出信息进行简单的处理，即可获得足够的信息。当然，如果你希望获取更多的信息，如当前系统使用内存最多的进程，可能还需要借助其他命令（如：top）才能得到。当然，不同的 OS 在输出值的处理方面可能会稍有区别，

- 进程数量：系统进程总数，或者某个用户下的进程数，都可以通过“ps”命令经过简单的处理来获得。如获取 mysql 用户下的进程总数：

  ```
  ps -ef | awk '{print $1}' | grep "mysql" | grep -v "grep" | wc -l
  ```

  如要获得更为详细的某个或者某类进程的信息，同样可以根据上述类似命令得到。

18.2.2 数据库健康状态信息

除了主机的状态信息之外，MySQL Server 自身也有很多的状态信息需要监控。下面就详细介绍一下 MySQL Server 需要监控的内容以及监控方法。

- 服务端口（3306）：MySQL 端口是一个必不可少的监控项，因为这直接反应了 MySQL Server 是否能够正常为外部请求提供服务。有些时候从主机层面来检查 MySQL 可能很正常，可外部应用却无法通过 TCP/IP 连接上 MySQL。产生这种现象的原因可能有多种，如网络防火墙的问题，网络连接问题，MySQL Server 所在主机的网络设置问题，以及 MySQL 本身问题都可能造成上述现象。
  服务端口状态的监控和主机网络连接的监控同样非常简单，只需要对 3306 端口进行 telnet 尝试即可。通过对 3306 端口的 telnet 尝试，同时还可以完成对主机网络状况的监控。

- socket 文件：对于有些环境来说，socket 的状态监控可能并不如网络服务端口的监控重要，因为很多 MySQL Server 的连接并不会通过本地 socket 进行连接。但是也有不少小型应用（或者某些特殊的应用）还是和 MySQL 数据库处于同一台主机上，并通过本地 socket 连接。另外，不少本地被动监控的信息采集脚本也是通过本地 socket 来连接 MySQL Server。
  本地 socket 的监控最好是通过本地 socket 的实际连接尝试的方式，虽然也可以通过检测 socket 文件是否存在来监控，但是即使 socket 文件确实存在，也并不一定就可以确保能够通过 socket 正常登录。毕竟，在某些异常情况下，socket 文件存在并不代表就可以正常使用。

- mysqld 和 mysqld_safe 进程：mysqld 进程是 MySQL Server 最核心的进程。mysqld 进程 crash 或者出现异常，MySQL Server 基本上也就无法正常提供服务了。当然，如果我们是通过 mysqld_safe 来启动 MySQL Server，则 mysqld_safe 会帮助我们来监控 mysqld 进程的状态，当 mysqld 进程 crash 之后，mysqld_safe 会马上帮助我们重启 mysqld 进程。但前提是我们必须通过 mysqld_safe 来启动 MySQL Server，这也是 MySQL AB 强烈推荐的做法。如果我

们通过 mysqld_safe 来启动 MySQL Server，那么我们也必须对 mysqld_safe 进程进行监控。
无论是 mysqld 还是 mysqld_safe 进程的监控，都可以通过 ps 命令来得到实现:

```
ps -ef | grep "mysqld_safe" | grep -v "grep"
和
ps -ef | grep "mysqld" | grep -v "mysqld_safe"| grep -v "grep"
```

● Error log：Error log 的监控目的主要是即时检测 MySQL Server 运行过程中发生的各种错误，如连接异常，系统 bug 等。
Error log 的监控和系统软硬件状态的监控一样，都是通过对文本文件内容的监控来实现的。同样使用文本监控软件，通过配置各种错误信息的匹配规则来捕获日志文件中的错误信息。

● 复制状态：如果我们的 MySQL 数据库环境使用了 MySQL Replication，就必须增加对 Slave 复制状态的监控。对 Slave 的复制状态的监控包括对 IO 线程和 SQL 线程二者的运行状态的监控。当然，如果希望能够监控 Replication 更多的信息，如两个线程各自运行的进度等，同样可以在 Slave 节点上执行相应命令轻松得到，如下：

```
sky@localhost : (none) 04:30:38> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State: Connecting to master
                  Master_Host: 10.0.77.10
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000001
          Read_Master_Log_Pos: 196
               Relay_Log_File: mysql-relay-bin.000001
                Relay_Log_Pos: 4
        Relay_Master_Log_File: mysql-bin.000001
             Slave_IO_Running: No
            Slave_SQL_Running: Yes
              Replicate_Do_DB: example,abc
          Replicate_Ignore_DB: mysql,test
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 196
```

```
                Relay_Log_Space: 106
                Until_Condition: None
                 Until_Log_File:
                  Until_Log_Pos: 0
             Master_SSL_Allowed: No
             Master_SSL_CA_File:
             Master_SSL_CA_Path:
                Master_SSL_Cert:
              Master_SSL_Cipher:
                 Master_SSL_Key:
          Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
                  Last_IO_Errno: 0
                  Last_IO_Error:
                 Last_SQL_Errno: 0
                 Last_SQL_Error:
```

通过如上命令，我们可以获取获取关于 Replication 的各种信息，不论是复制的监控状况，还是复制的进度，都可以很容易的获取。上面输出的各项内容中，最重要的内容就是 Slave_IO_Running 和 Slave_SQL_Running 这两项，分别代表了 IO 线程和 SQL 线程的运行状态，如上面的输出就表明 SQL 线程运行正常，但是 IO 线程并没有运行。其他各项的详细解释，请参考本书附录。



18.3 性能状态监控

性能状态的监控和健康状态的监控有一定的区别，他所反应出来的主要是系统当前提供服务的响应速度，影响的更多是客户满意度。性能状态的持续恶化，则可能升级为系统不可用，也就是升级为健康状态的异常。如系统的过度负载，可能导致系统的 crash。性能状态的监控同样可以分为主机和数据库层面。

18.3.1 主机性能状态

主机性能状态主要表现在三个方面：CPU、IO 以及网络，可以通过以下五个监控量来监控。

- 系统 load 值：系统 load 所包含的最关键含义是 CPU 运行等待的数量，也就是侧面反应了 CPU 的繁忙程度。只不过 load 值并不直接等于等待队列中的进程数量。

load 值的监控也非常简单，通过运行 uptime 命令即可获得当前时间之前 1 秒、5 秒和 15 秒的 load 平均值。

```
sky@sky:~$ uptime
17:27:44 up  4:12,  3 users,  load average: 0.87, 0.66, 0.61
```

如上面输出内容中的“load average: 0.87, 0.66, 0.61”中的三个数字，分别代表了 1 秒、5 秒和 15 秒的 load 平均值。
一般来说，load 值在不超过系统物理 cpu 数目（或者 cpu 总核数）之前，系统不会有太大问题。

- CPU 使用率：CPU 使用率和系统 load 值一样，从另一个角度反应出 CPU 的总体繁忙程度，只不过可以反应出更为详细的信息，如当前空闲 的 CPU 比率，系统占用的 CPU 比率，用户进程占用的 CPU 比率，处于 IO 等待的 CPU 比率等。
  CPU 使用率可以通过多种方法来获取，最为常用的方法是使用命令 top 和 vmstat 来获取 。当然，不同的 OS 系统上的 top 和 vmstat 的输出可能有些许不同，且输出格式也可能各不一样，如 Linux 下的 top 包含 IO 等待的 CPU 占用律，但是 Solaris 下的 top 就不包含，各位读者朋友请根据自己的 OS 环境进行相应的处理。

- 磁盘 IO 量：磁盘 IO 量直接反应出了系统磁盘繁忙程度，对于数据库这种以 IO 操作为主的系统来说，IO 的负载将直接影响到系统的整体响应速度。
  磁盘 IO 量同样也可以通过 vmstat 来获取，当然，我们还可以通过 iostat 来获得更为详细的系统 IO 信息。包括各个磁盘设备的 iops，每秒吞吐量等。

- swap 进出量：swap 的使用主要表现了系统在物理内存不够的情况下使用虚拟内存的情况。当然，有些时候即使系统还有足够物理内存的时候，也可能出现使用 swap 的情况，这主要是由系统内核中的内存管理部分来决定。如果希望系统完全不使用 swap，最直接的办法就是关闭 swap。当然，前提条件是我们必须要有足够的物理内存，否则很可能会出现内存不足的错误，严重的时候可能会造成系统 crash。
  swap 使用量的总体情况可以通过 free 命令获得，但 free 命令只能获得当前系统 swap 的总体使用量。如果希望获得实时的 swap 使用变化，还是得依赖 vmstat 来得到。vmstat 输出信息中包含了每秒 swap 的进出量，当然，不同的 OS 输出可能存在一定的差异。

- 网络流量：作为数据库系统，网络流量也是一个不容忽视的监控点。毕竟数据库系统的数据进出比普通服务器的量还是要大很多的。不论是总体吞吐量还是网络 iops，都需要给予一定的关注。当然，一般非数据仓库类型的数据库，网络流量成为瓶颈的可能性还是比较小的。
  网络流量的监控很少有系统自带的命令可以直接完成，而需要自行编写脚本或者通过一些第三方软件来获取数据。当然，通过对网络交换机的监控，也可以获得非常详细的网络流量信息。自行编写脚本可以通过调用 ifconfig 命令来计算出基本准确的网络流量信息。第三方软件如 ifstat、iftop 和 nload 等则是需要另外安装的监控 linux 下网络流量第三方软件，三者各有特点，读者朋友可以根据三款软

件官方介绍的功能特点自行选择。

18.3.2 数据库性能状态

MySQL 数据库的性能状态监控点非常之多，其中很多量都是我们不能忽视的必须监控的量，且 90% 以上的内容可以在连接上 MySQL Server 后执行“SHOW /*!50000 GLOBAL */ STATUS” 以及 “SHOW /*!50000 GLOBAL */ VARIABLES” 的输出值获得。需要注意的是上述命令所获得状态值实际上是累计值，所以如果要计算(单位/某个)时间段内的变化量还需要稍加处理，可以在附录中找到两个命令输出值的详细说明。下面看看几项需要重点关注的性能状态：

- QPS（每秒 Query 量）：这里的 QPS 实际上是指 MySQL Server 每秒执行的 Query 总量，在 MySQL 5.1.30 及以下版本可以通过 Questions 状态值每秒内的变化量来近似表示，而从 MySQL 5.1.31 开始，则可以通过 Queries 来表示。Queries 是在 MySQL 5.1.31 才新增的状态变量。主要解决的问题就是 Questions 状态变量并没有记录存储过程中所执行的 Query（当然，在无存储过程的老版本 MySQL 中则不存在这个区别），而 Queries 状态变量则会记录。二者获取方式：

  QPS = Questions(or Queries) / Seconds

  获取所需状态变量值：
  SHOW /*!50000 GLOBAL */ STATUS LIKE 'Questions'
  SHOW /*!50000 GLOBAL */ STATUS LIKE 'Queries'

  这里的 Seconds 是指累计出上述两个状态变量值的时间长度，后面用到的地方也代表同样的意思。

- TPS（每秒事务量）：在 MySQL Server 中并没有直接事务计数器，我们只能通过回滚和提交计数器来计算出系统的事务量。所以，我们需要通过以下方式来得到客户端应用程序所请求的 TPS 值：

  TPS = (Com_commit + Com_rollback) / Seconds

  如果我们还使用了分布式事务，那么还需要将 Com_xa_commit 和 Com_xa_rollback 两个状态变量的值加上。

- Key Buffer 命中率：Key Buffer 命中率代表了 MyISAM 类型表的索引的 Cache 命中率。该命中率的大小将直接影响 MyISAM 类型表的读写性能。Key Buffer 命中率实际上包括读命中率和写命中率两种，MySQL 中并没有直接给出这两个命中率的值，但是可以通过如下方式计算出来：

  key_buffer_read_hits = (1 - Key_reads / Key_read_requests) * 100%

key_buffer_write_hits= (1 - Key_writes / Key_write_requests) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 07:44:10> SHOW /*!50000 GLOBAL */ STATUS
    -> LIKE 'Key%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
    ... ...
| Key_read_requests       | 10    |
| Key_reads               | 4     |
| Key_write_requests      | 0     |
| Key_writes              | 0     |
+-------------------------+-------+
```

通过这两个计算公式，我们很容易就可以得出系统当前 Key Buffer 的使用情况

- Innodb Buffer 命中率：这里 Innodb Buffer 所指的是 innodb_buffer_pool，也就是用来缓存 Innodb 类型表的数据和索引的内存空间。类似 Key buffer，我们同样可以根据 MySQL Server 提供的相应状态值计算出其命中率：
innodb_buffer_read_hits=(1-Innodb_buffer_pool_reads/Innodb_buffer_pool_read_requests) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 08:25:14> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Innodb_buffer_pool_read%';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
    ... ...
| Innodb_buffer_pool_read_requests  | 5367  |
| Innodb_buffer_pool_reads          | 507   |
+-----------------------------------+-------+
```

- Query Cache 命中率：如果我们使用了 Query Cache，那么对 Query Cache 命中率进行监控也是有必要的，因为他可能告诉我们是否在正确的使用 Query Cache。Query Cache 命中率的计算方式如下：

Query_cache_hits= (Qcache_hits / (Qcache_hits + Qcache_inserts)) * 100%

获取所需状态变量值：

```
sky@localhost : (none) 08:32:01> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Qcache%';
+-------------------------+-------+
```

```
| Variable_name           | Value |
+-------------------------+-------+
    ... ...
| Qcache_hits             | 0     |
| Qcache_inserts          | 0     |
    ... ...
+-------------------------+-------+
```

- Table Cache 状态量：Table Cache 的当前状态量可以帮助我们判断系统参数 table_open_cache 的设置是否合理。如果状态变量 Open_tables 与 Opened_tables 之间的比率过低，则代表 Table Cache 设置过小，个人认为该值处于 80% 左右比较合适。注意，这个值并不是准确的 Table Cache 命中率。

  获取所需状态变量值：

```
sky@localhost : (none) 08:52:00> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Open%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
    ... ...
| Open_tables              | 51    |
    ... ...
| Opened_tables            | 61    |
+--------------------------+-------+
```

- Thread Cache 命中率：Thread Cache 命中率能够直接反应出我们的系统参数 thread_cache_size 设置的是否合理。一个合理的 thread_cache_size 参数能够节约大量创建新连接时所需要消耗的资源。Thread Cache 命中率计算方式如下：

  Thread_cache_hits = (1 - Threads_created / Connections) * 100%

  获取所需状态变量值：

```
sky@localhost : (none) 08:57:16> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Thread%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
    ... ...
| Threads_created   | 3     |
    ... ...
+-------------------+-------+
4 rows in set (0.01 sec)

sky@localhost : (none) 09:01:33> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Connections';
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Connections   | 11    |
+---------------+-------+
```

正常来说，Thread Cache 命中率要在 90% 以上才算比较合理。

- 锁定状态：锁定状态包括表锁和行锁两种，我们可以通过系统状态变量获得锁定总次数，锁定造成其他线程等待的次数，以及锁定等待时间信息。

```
sky@localhost : (none) 09:01:44> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE '%lock%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
    ... ...
| Innodb_row_lock_current_waits | 0     |
| Innodb_row_lock_time          | 0     |
| Innodb_row_lock_time_avg      | 0     |
| Innodb_row_lock_time_max      | 0     |
| Innodb_row_lock_waits         | 0     |
    ... ...
| Table_locks_immediate         | 44    |
| Table_locks_waited            | 0     |
+-------------------------------+-------+
```

通过上述系统变量，我们可以得出表锁总次数，其中造成其他现线程等待的次数。同时还可以得到非常详细的行锁信息，如行锁总次数，行锁总时间，每次行锁等待时间，行锁造成最大等待时间以及当前等待行锁的线程数。通过对这些量的监控，我们可以清晰的了解到系统整体的锁定是否严重。如当 Table_locks_waited 与 Table_locks_immediate 的比值较大，则说明我们的表锁造成的阻塞比较严重，可能需要调整 Query 语句，或者更改存储引擎，亦或者需要调整业务逻辑。当然，具体改善方式必须根据实际场景来判断。而 Innodb_row_lock_waits 较大，则说明 Innodb 的行锁也比较严重，且影响了其他线程的正常处理。同样需要查找出原因并解决。造成 Innodb 行锁严重的原因可能是 Query 语句所利用的索引不够合理（Innodb 行锁是基于索引来锁定的），造成间隙锁过大。也可能是系统本身处理能力有限，则需要从其他方面（如硬件设备）来考虑解决。

- 复制延时量：复制延时量将直接影响了 Slave 数据库处于不一致状态的时间长短。如果我们是通过 Slave 来提供读服务，就不得不重视这个延时量。我们可以通过在 Slave 节点上执行“SHOW SLAVE STATUS”命令，取 Seconds_Behind_Master 项的值来了解 Slave 当前的延时量（单位：秒）。当然，该值的准确性依赖于复制是

否处于正常状态。每个环境下的 Slave 所允许的延时长短与具体环境有关，所以复制延时多长时间是合理的，只能由读者朋友根据各自实际的应用环境来判断。

- Tmp table 状况：Tmp Table 的状况主要是用于监控 MySQL 使用临时表的量是否过多，是否有临时表过大而不得不从内存中换出到磁盘文件上。临时表使用状态信息可以通过如下方式获得：

```
sky@localhost : (none) 09:27:28> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Created_tmp%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 1     |
    ... ...
| Created_tmp_tables      | 46    |
+-------------------------+-------+
```

从上面的状态信息可以了解到系统使用了 46 次临时表，其中有 1 次临时表比较大，无法在内存中完成，而不得不使用到磁盘文件。如果 Created_tmp_tables 非常大，则可能是系统中排序操作过多，或者是表连接方式不是很优化。而如果是 Created_tmp_disk_tables 与 Created_tmp_tables 的比率过高，如超过 10%，则我们需要考虑是否 tmp_table_size 这个系统参数所设置的足够大。当然，如果系统内存有限，也就没有太多好的解决办法了。

- Binlog Cache 使用状况：Binlog Cache 用于存放还未写入磁盘的 Binlog 信息。相关状态变量如下：

```
sky@localhost : (none) 09:40:38> SHOW /*!50000 GLOBAL*/ STATUS
    -> LIKE 'Binlog_cache%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Binlog_cache_disk_use | 0     |
| Binlog_cache_use      | 0     |
+-----------------------+-------+
```

如果 Binlog_cache_disk_use 值不为 0，则说明 Binlog Cache 大小可能不够，建议增加 binlog_cache_size 系统参数大小。

- Innodb_log_waits 量：Innodb_log_waits 状态变量直接反应出 Innodb Log Buffer 空间不足造成等待的次数。

```
sky@localhost : (none) 09:43:53> SHOW /*!50000 GLOBAL*/ STATUS
-> LIKE 'Innodb_log_waits';
```

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| Innodb_log_waits | 0     |
+------------------+-------+
```

该变量值发生的频率将直接影响系统的写入性能，所以当该值达到每秒 1 次时就该增加 系统参数 innodb_log_buffer_size 的值，毕竟这是一个系统共用的缓存，适当增加并不会造成内存不足的问题。

上面这些监控量只是我个人认为比较重要的一些 MySQL 性能监控量，各位读者朋友还可以根据各自的需要，通过 MySQL 所提供的系统状态变量增加其他监控内容。

18.4 常用开源监控软件

前面已经介绍过了监控系统的设计思路，也分析了我们需要关注的部分健康状态和性能状态监控点，这一节再介绍几种常用的第三方监控软件，为大家提供一点搭建监控系统的思路。当然，推荐原则仍然是以开源（或免费）产品为主。当然，前提是我们暂时没有自行开发一套监控系统的打算，而希望通过一些开源的软件工具来实现。

在介绍监控软件之前，有一个软件是我不得不提的：RRDTool。RRDTool 全称为 Round Robin Database Tool，也就是环状循环数据库工具，在多种监控软件中充当非常重要的数据存储角色。所谓环状循环数据库，就是数据的存储方式类似于在一个环形空间中，没有确切的起始位置和结束位置。当写完一圈之后，新的数据会覆盖老数据。当然，由于环状循环数据库所存放的数据大多都是存放用于统计方面的数据，而且可以通过一些加和平均之类的统计算法通过老数据得出相应的统计结果。当老数据需要被覆盖的时候，他早已失去实际价值了。所以，RRDTool 在很多监控工具中都被用来存放各种性能数据，然后根据这些数据画出相应的趋势图，以及不同时间段内的平均值曲线。如果有哪位读者朋友有兴趣自行开发一个监控工具，RRDTool 同样可以作为您用来存放相关数据非常合适的选择。不过，这里有一点需要注意的是他只能存放数字。

1、 Nagios

Nagois 是一个非常著名的运行在 Linux/Unix 上的对 IT 设备或服务的运行状态进行监控的软件。实际上，我个人觉得称其为一个监控平台可能更为合适，因为它不仅仅自带了丰富的监控工具，同时还支持用户自己编写各种各样的监控脚本以插件形式嵌入其中。

Nagios 自带的监控功能不仅包含与主机资源相关的 CPU 负载，磁盘使用等，还包括了网络相关的服务，如 SMTP、POP3、HTTP、NNTP、PING 等。当然，Nagios 流行的另外一个重要原因是其简单的插件设计允许用户可以非擦方便地开发自己需要的服务检查。

对于大多数的监控场景来说，Nagios 的现有功能都能够满足。不论是主动检测，还是被动监控，都可以很好的实现。对于不同重要级别的监控点，可以分别设置不同的数据采集频率。对于异常的处理，可以设定为邮件、Web 以及其他自定义的异常通知机制（如手机短

信或者 IM 工具通知)。不仅如此，Nagios 还可以设置报警前的异常出现次数，如连续多少次访问超时之后再发出警告信息。而当我们需要进行正常维护的时候，还可以通过设置一个暂时忽略异常的 DownTime 时间段。

Nagios 分为客户端与服务器端两部分，客户端实际上就是大量的 Nagios 监控插件，负责采集监控点的各种数据，而服务器端则是配置管理、数据分析、告警发送、用户自助管理以及相关信息展示等功能。服务器端的Web展示界面的功能也非常强大，可以非常准且的展示出当前各个监控点的状态，上一次检测时间等信息。同时还能根据监控点追溯一定的历史记录信息。

当然，Nagios 也有一个缺点，那就是目前他仅仅支持健康状态数据的采集分析，而不支持通过采集性能数据来绘制性能趋势曲线图。当然，这可以结合其他的软件工具共同完成这一工作。

如需更为详细的 Nagios 的搭建使用手册，请各位读者朋友前往其官方网站获取更为权威的信息：http://www.nagios.org/docs

2、MRTG

MRTG 应该算是一款比较老牌的监控软件了，功能比较简单， 最初是为了监控网络链路流量而产生的。但是经过原开发者的不断改造，以及网友们的集体智慧，其应用场景已经远远超出了网络链路流量监控。

MRTG 的实现原理其实很简单，他通过 snmp 协议，调用监控设备上面相应的脚本，获取需要的返回值，然后通过 RRDTool 保存起来。然后再通过 RRDTool 将保存的数据进行相应的统计平均计算，最后画成图片，通过 html 的形式展现给前端浏览者。

对于 MRTG 来说，它不需要知道我们监控的到底是什么数据，只需要告诉他到底什么时候该调用什么脚本来取得监控返回值，同时配置好存放位置，即可完成一个监控项的监控配置。对于我们来说，最为重要的是写好取得监控值的脚本，设定好 MRTG 的采集频率，然后就是查看各种监控数据的曲线图了。

更为详细的 MRTG 使用及搭建方法，请至官方网站(http://oss.oetiker.ch/mrtg) 上查找相应文档。

3、 Cacti

Cacti 和 Nagios 最大的区别在于前者具有非常强大的数据采集、存储以及展现功能，但在告警管理这一块稍弱于后者。其开发语言是PHP,配置存储在 MySQL 数据库中,数据采集同样利用了 SNMP 协议，采集数据的存储则利用了本节最前面介绍的 RRDTool。

在数据的绘图展现方面，相对于其他有些监控软件来说，也有一定的优势，如单个图上面可以有无限多个数据项共存，而不像 MRTG 每张图片上只能有两个数据项的曲线。

除了非常强大的数据灰土展现功能之外，Cacti 另外一个比较有吸引力的功能就是可以

通过用户管理，设定多种权限角色，让更多的用户自行定义维护各自的监控配置项。这个特性对于监控设备（或服务）涉及到多个部门的很多人的应用场景下，是非常有用的。

当然，除了上面的这几个比较有特点特性之外，Cacti 还存在很多很多其他的特性。大家可以从 Cacti 官方文档获得更详细的信息：http://www.cacti.net/documentation.php。

不同的监控需求，可以采用不同的监控软件来实现，如我个人的很多环境就同时使用了 Nagios 来实现健康状况的监控和告警。而性能数据的采集与绘图则利用了 MRTG 和 RRDTool 来实现。各位读者朋友完全可以根据自己的需求来决定如何搭建一个更为合适的监控系统，来帮助大家更进一步提高系统的可用性。

18.4 小结

系统的监控在很多环境中都没有得到足够的重视，可其实际意义却非常大。虽然监控系统本身并不能让系统运行的更好，却能够给在系统出现问题之后的第一时间通知我们，缩短了发现问题的时间，也间接提高了系统的可用性。甚至可以根据监控所收集的各种性能数据，让我们提前发现系统的性能问题，防范于未然。

本章通过对 MySQL 数据库环境的健康状态、性能状态以及相应的监控方式的分析，完成了搭建一个高可用数据库环境的最后一步，希望能够对各位读者朋友有一点帮助。